

Рис. 2. Конвергенция:
а — акула; б — ихтиозавр; в — дельфин

фу 11 кции находит подтверждение в морфологии и физиологии. В настоя-
11 исе время физиология как наука имеет множество направлений, базирую-
щихся на учении И. П. Павлова о нервной системе, объединяющей орга-
низм в единое целое.

Эмбриология (от греч. *embryon* — зародыш) — наука о зародышевом развитии животных. Сравнительная эмбриология позволяет установить закономерности в развитии зародышей разных животных, понять взаимосвязь онтогенеза и филогенеза. Пройденные исторические этапы в эволюционном развитии животных в какой-то степени проявляются у современных видов на эмбриональных стадиях их развития. Менно в раннем периоде развития у зародышей разных видов животных наблюдается наибольшее сходство. Такая закономерность была сформулирована К. Бэрром как закон зародышевого сходства. Сущность этого закона заключается в том, что в эмбриональном периоде развития раньше других закладываются признаки типа, к которому принадлежит животное, затем признаки класса, позднее признаки отряда и, наконец, оформляются признаки вида. Эта закономерность повторения в онтогенезе стадий филогенеза была обобщена Э. Геккелем в форме биогенетического закона: онтогенез есть краткое повторение филогенеза.

Экология (от греч. *oikos* — жилище, местообитание) изучает животных в связи с местом и условиями их жизни, закономерности во взаимосвязях организмов со средой обитания. Одной из задач экологии является исследование приспособительных черт в строении животных, в их жизненных направлениях, в поведении. Появляется новая наука — **этология** (от греч. *Sthos* — характер, нрав), которая позволяет раскрыть многие закономерности жизнедеятельности сообществ животных, взаимоотношения отдельных видов, и в частности конкуренцию, паразитизм и т. п. Данные экологии и этологии ценны не только для разра-

ботки теоретических проблем общебиологического характера, но несут и весьма ценную информацию для животноводов-практиков, охотников, работников зоопарков, заповедников и т. д.

Зоогеография изучает закономерности географического распространения животных на нашей планете. Исследования в области зоогеографии, экологии и этологии создают предпосылки к практическому использованию всех полученных результатов для обоснования мер по реконструкции фауны, для научного подхода к разработке и осуществлению проектов, затрагивающих целые регионы страны (строительство авто- и железнодорожных магистралей, переброска рек и т. п.).

Палеонтология (от греч. *palaios* — древний) изучает вымерших животных (палеозоология) и растений (палеоботаника), восстанавливает по окаменелостям облик вымерших животных и растений. Ж. Кювье дал научное обоснование распределения окаменелостей по слоям осадочных пластов Земли разного геологического возраста. В. О. Ковалевский призывал палеонтологов не только констатировать различия в фаунах разных эпох, но и искать среди ископаемых остатков сходство между животными смежных напластований.

Филогенетика (от греч. *phylon* — племя, род; *genesis* — происхождение) занимается изучением исторических связей в мире животных и растений. В результате изучения родственных отношений животных было построено родословное древо животного мира.

Систематика (от греч. *systematikos* — упорядоченный) на основе данных всех зоологических наук разрабатывает классификацию животных и естественную систему животного мира, отражающую родственные связи различных групп животных. Естественная система базируется на том, что все организмы постоянно развиваются. Эта система предназначена отражать генеалогические связи организмов, ибо для объединения животных в ту или иную категорию основанием служит степень сходства как отражение родства. К одному виду относят животных, наиболее близко родственных (что и объясняет их морфологическое сходство), свободно скрещивающихся между собой с получением плодовитого потомства и имеющих общее географическое распространение. Видовые признаки стойкие и отличаются исключительной консервативностью.

В результате внутривидовой изменчивости образуются подвиды и разновидности. Общеизвестны различия в окраске, опущенности, в размерах и т. п. у животных одного вида, но обитающих в разных географических зонах. Особенно большая пластичность свойственна моллюскам и насекомым.

Особенности организма животных. Эволюция живой природы на Земле привела к образованию животных и растений. Важное различие между ними заключается в характере обмена веществ, который обусловлен типом питания. Если растения в большинстве своем автотрофные организмы, то животные, как правило, гетеротрофные организмы. В темноте растения погибают, а животные способны жить.

Большинство животных ведут активный образ жизни, свободно перемещаясь в пространстве или совершая разнообразные движения.

Гас гения обычно неподвижны. Клетки животных не имеют плотных неточных стенок, построенных из целлюлозных волокон, и не содержат вакуолей с клеточным соком, которые свойственны клеткам растений! Но провести резкую границу между животными и растениями невозможно; особенно это трудно сделать для низших их форм, сохраняющих черты, общие для двух царств природы. Одна из кардинальных ощущений черт животных и растений — их клеточное строение.

Клетка. Клетка представляет собой основную структурно-функциональную единицу всех живых организмов. В теле многоклеточных животных клетки дифференцированы в зависимости от выполняемых ими функций, что обусловливает их различия не только по размерам, но также по форме и строению. Все клетки организма взаимосвязаны и взаимодействуют друг с другом; такая связь и взаимодействие осуществляются через их плазматическую мембрану. В клетках протекают процессы обмена веществ.

Типичная животная клетка содержит цитоплазму, ядро (или ядра) и различные органоиды, или органеллы; сама клетка ограничена наружной мембраной, которую называют плазматической мембраной или плазмалеммой (рис. 3).

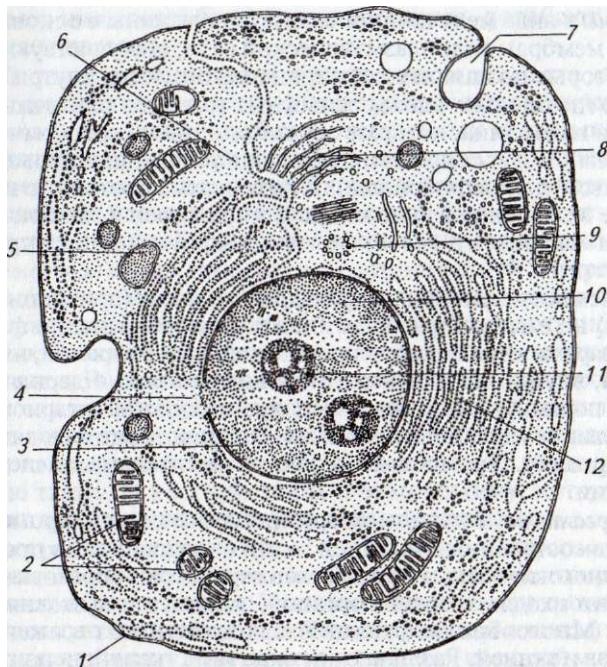


Рис. 3. Строение клетки эукариот:
1 — плазматическая мембрана; 2 — митохондрии; 3 — кариоплазма; 4 — ядерная оболочка;
5 — лизосома; 6 — цитоплазма; 7 — пиноцитозный пузырек; 8 — аппарат Гольджи;
9 — центриоли; 10 — ядро; 11 — ядрышко; 12 — эндоплазматическая сеть

Плазматическая мембрана представляет собой сложный белко-во-липидный комплекс. Она очень тонка и не только защищает клетку от внешних воздействий, но и участвует в обмене веществами между клеткой и окружающей средой.

Цитоплазма — это сложная коллоидная система, в которой находятся структурные образования, такие как митохондрии, эндоплазматическая сеть, аппарат Гольджи, рибосомы и растворенные вещества.

Митохондрии, имеющие вид мелких удлиненных телец, служат энергетическими центрами клетки, регулирующими биохимические реакции превращения энергии.

Эндоплазматическая сеть, штэндоплазматический ретикулум, представляет собой систему тончайших трубочек, пронизывающих всю цитоплазму клетки, и пузырьков. По тончайшим каналцам эндоплазматической сети осуществляется внутриклеточный обмен веществ.

Аппарат Гольджи близок по своему строению к строению эндоплазматической сети и служит для временного хранения продуктов внутриклеточного синтеза (в основном гормонов и ферментов) и передачи их через эндоплазматическую сеть для вовлечения в обменные процессы всего организма.

Рибосомы в виде мельчайших зерен расположены в основном на поверхности мембран эндоплазматической сети; они участвуют в синтезе белков, которые по каналам сети транспортируются внутри клетки.

Часто в цитоплазме клеток животных имеются различные тончайшие нити и волоконца, которые служат опорным каркасом клеток (тонофибриллы), могут сокращаться (миофибриллы) или проводить нервные импульсы (нейрофибриллы). В цитоплазме постоянно наблюдают временные включения в виде капелек жира, зерен и глыбок резервных белков, пигментов и т. д., которые возникают и исчезают в процессе обмена веществ.

Лишь немногие специализированные клетки (эритроциты млекопитающих) не имеют ядра. *Ядра* клеток разнообразны по форме и величине. Снаружи ядро ограничено двухслойной мембраной, или ядерной мембраной, внутри которой находится кариоплазма. Ядерная мембрана пронизана порами, от которых в сторону цитоплазмы и кариоплазмы отходят небольшие тончайшие каналцы. В кариоплазме находятся хромосомы и ядрышко. Хромосомы являются носителями наследственной информации.

Ткани организма. В животном организме все клетки, кроме половых, находятся в составе тканей. Ткани — это сложившиеся в процессе филогенеза многоклеточных организмов структуры, образованные клетками. Ткани входят в состав органов, участвуя в выполняемых ими функциях. Многообразие функций тела животного отражено в строении органов и тканей. Различают четыре типа тканей: нервную, эпителиальную, соединительную и мышечную.

Нервная ткань воспринимает и передает раздражения, поступающие как из внешней, так и из внутренней среды организма. Раздражимость — одно из свойств, характеризующих живую материю. Нервная

II .in I, состоит из нервных клеток, или нейронов, клеток глии и межклеточного вещества. В зависимости от выполняемой функции нейроны делятся на чувствительные и двигательные. Каждый нейрон имеет • •/•nil или несколько отростков. Короткие разветвленные отростки называются *дendritами*, а один длинный отросток называется *нейритом* и *in аксоном*. Концевые разветвления аксона чувствительной клетки принимают раздражение и называются *рецепторами*. Их много на поверхности тела и во внутренних органах. От рецептора возбуждение передается по аксону к телу нейрона, а затем по его дендритам — к дендритам двигательного нейрона. В результате возникает соответствующая реакция — двигательная, секреторная и т. п.

Эпителиальная ткань представляет собой пластины клеток, плотно прилегающих друг к другу и соединенных межклеточными контактами. Под эпителиальным пластом располагается слой межклеточного вещества, называемого базальной мембраной. Эпителий бывает однолисточным (его клетки лежат в один ряд) и многослойным (его клетки располагаются в несколько рядов). Для эпителия характерна высокая способность к регенерации, поскольку его клетки из-за своего положения (на поверхности тела или на внутренней поверхности полых органов, например пищевода) быстро изнашиваются, погибают и должны заменяться новыми.

Кожный эпителий находится в постоянном контакте с внешней средой, что определяет его специфику у разных видов животных. Например, кожа рыб обильно снабжена слизистыми клетками, а у насекомых эпителий пропитан хитиновым веществом, защищающим тело от высыхания. Вместе с тем энтодермальный эпителий кишечника выложен в один слой клетками цилиндрической формы, через этот эпителий осуществляется избирательное всасывание переваренной пищи.

Соединительная ткань относится к системе тканей внутренней среды. Для соединительной ткани характерно наличие большого количества межклеточного вещества и сравнительно небольшого числа клеток. И кости межклеточное вещество плотное, а в крови — жидкое. Соединительная ткань выполняет многообразные функции: трофическую (связанную с питанием организма), опорную, защитную и др. В зависимости от выполняемых функций соединительную ткань делят на: 1) собственно соединительную ткань, 2) жировую ткань, 3) кровь, 4) хрящевую ткань, 5) костную ткань и 6) мезенхиму, которую можно рассматривать как соединительную ткань зародыша.

В состав крови входят ее жидкая часть (плазма) и форменные элементы — клетки крови. Форменные элементы у животных разных типов существенно различаются. У позвоночных это эритроциты (красные кровяные клетки), лейкоциты (белые кровяные клетки) и тромбоциты. Главным органом кроветворения является костный мозг. Особая роль принадлежит эритроцитам, поглощающим кислород и транспортирующим его к тканям. Лейкоциты наиболее разнообразны у позвоночных по форме и по выполняемым функциям, среди которых важное место принадлежит нейтрализации ядов и чужеродных тел.

Наряду с кровеносной системой у позвоночных имеется *лимфатическая система*. Представлена она лимфатическими узлами и лимфатическими сосудами. Лимфоциты, циркулирующие по лимфатической системе, играют важную роль в защитной функции организма.

Ретикулярная ткань в виде рыхлого скопления звездчатых клеток является основой селезенки. Наличие большого количества лимфоцитов способствует процессам фагоцитоза, определяя защитные функции этой ткани.

Рыхлая волокнистая соединительная ткань входит в состав многих органов и подкожной клетчатки. Плотная волокнистая соединительная ткань составляет основу связок (эластиновые волокна) и нижнего слоя кожи (коллагеновые волокна).

Хрящевая ткань входит в состав скелета позвоночных и ряда беспозвоночных животных. Она составляет основу ушных раковин, образует межпозвоночные диски. Из костной ткани сформированы кости. Костная ткань состоит из клеток и межклеточного вещества, построенного из коллагеновых волокон и аморфной массы, пропитанной минеральными солями. Костная ткань — это живая ткань; в ней находятся кровеносные сосуды и нервы. Кости в организме служат депо кальция, фосфора и других минеральных элементов.

Мышечная ткань. Мышцы делятся на гладкие и поперечно-полосатые. Основу мышц составляют тончайшие мышечные волоконца — миофибриллы. Гладкие мышцы характеризуются плавностью сокращения и расслабления. Они находятся во внутренних органах. Поперечно-полосатые мышцы способны совершать быстрые сокращения и выносить большую нагрузку. При этом скорость их сокращения может колебаться в значительных пределах. Поперечно-полосатые мышцы обычно прикреплены к костям наружного или внутреннего скелета и относятся к мышцам произвольного сокращения.

Из тканей формируются различные органы многоклеточного животного. Функционирование любого органа происходит в тесном взаимодействии со всеми другими органами, что свидетельствует о целостности всего организма. То же самое наблюдается и в деятельности различных органелл у одноклеточных животных; особенно четко это прослеживается у высших простейших — инфузорий. Каждый орган функционирует как неразрывная часть единого организма.

Особенности строения и функционирования различных органов изучаются в разделах систематики животных.

Размножение — это свойство живых организмов воспроизводить себе подобных особей. Животные размножаются бесполым и половым путем. *Бесполое размножение* характерно в основном для низших животных. При бесполом размножении от материнской особи либо отделяется часть ее тела, либо вся материнская особь делится на две или большее число частей. При этом каждая часть в дальнейшем развивается в самостоятельное животное. Существует несколько способов бесполого размножения: деление, почкование и шизогония.

При размножении *простым делением* ма ге р и некая особь делится на две одинаковые дочерние особи (рис. 4). У одних простейших (жгутиконосцы) тело делится п продольном направлении, у других (инфузории) — в поперечном, у третьих, облагающих шарообразной или изменчивой формой (например, амебы), деление может происходить в любом направлении.

При *почковании* на теле материнского организма образуется вырост — почка, которая постепенно приобретает форму и строение взрослой особи. После отделения (ограничения) от материнского организма новая (дочерняя) особь начинаетести самостоятельную жизнь. При почковании у многих видов кишечнополостных животных, ведущих сидячий образ жизни, в результате неполного отделения дочернего организма (почки) от материнского и сохранения связи с последним образуются колонии. При этом могут возникать целые коралловые «города» — рифы.

У ряда паразитических форм простейших наблюдается множественное деление — *шизогония*. В этом случае ядро материнского организма многократно делится и образуется многоядерный *шизонт*. Вокруг каждого ядра внутри шизонта обособляется участок цитоплазмы. Шизонт распадается (делится) на многочисленные мелкие дочерние особи — *мерашиты*. Такое множественное деление позволяет паразиту в очень короткий срок достичь высокой численности в организме своего хозяина.

Половое размножение свойственно всем типам животных. При размноженииовым путем новый организм развивается из зиготы, которая образуется в результате слияния женской (яйца) и мужской (спермии) половых клеток.

Спермий — мужская гаплоидная половая клетка — обычно состоит из головки, шейки и хвоста. Последний служит для передвижения спермии в жидкой или вязкой среде (рис. 5).

Яйцеклетка — женская половая клетка — имеет округлую форму и состоит из цитоплазмы и ядра. По своим размерам яйцеклетка пре-восходит спермий во много раз.

У некоторых животных наблюдается наружное оплодотворение, т. е. женские и мужские гаметы выделяются в воду, где и происходит их слияние. Другим животным свойственно внутреннее оплодотворение: спермии в составе спермы вводятся в половые пути самки, где и происходит их слияние с яйцеклеткой.

Имеют место случаи развития организмов из неоплодотворенных яйцеклеток. Такое размножение, называемое *партеногенетическим* или *девственным* размножением, часто встречается у членистоногих.

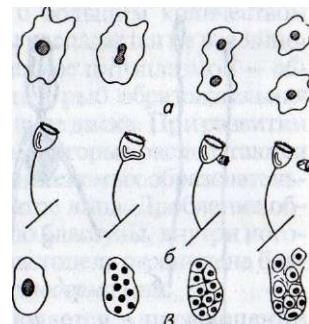


Рис. 4. Бесполое размножение одноклеточных животных:
а — деление амебы; б — почкование инфузории суворки;
в — шизогония малярийного плазмодия

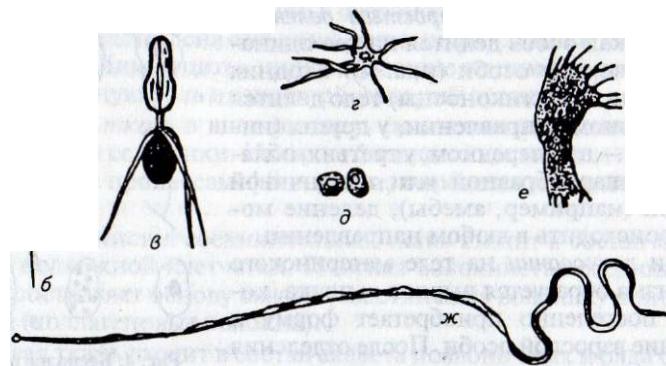


Рис. 5. Различные формы спермиев животных:
а — улитки; *б* — медузы; *в* — рака; *г* — круглого червя; *д, е* — низших раков; *ж* — жука

Начальные этапы развития многоклеточных животных. Развитие организма начинается с делений дробления зиготы (оплодотворенного яйца) на ряд клеток — *blastomeres*. Дробление бывает полным и неполным, что обусловлено количеством желтка в яйце. Полное дробление происходит тогда, когда в яйце содержится мало желтка и он распределен в цитоплазме относительно равномерно (такие яйца называют *гомолецитальными*). При полном дроблении все яйцо делится сначала на 2 blastomera, затем на 4, 8, 16, 32 и т. д. В результате такого полного дробления яйца образуется комочек из клеток — *морула*. В последующем в центре морулы возникает полость и морула превращается в однослоистый зародыш — *blastula* (рис. 6).

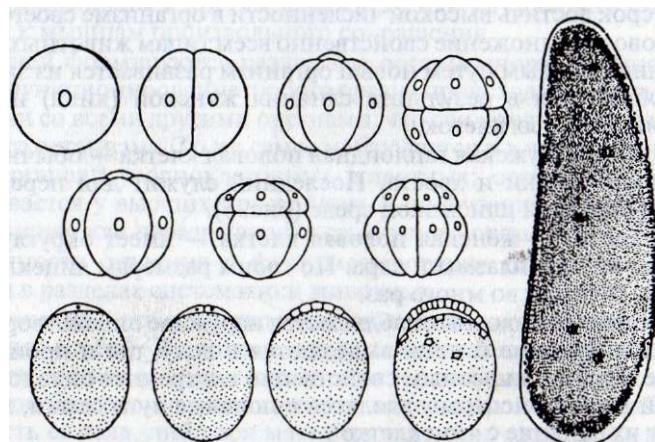


Рис. 6. Типы дробления яйца:
а — полное равномерное; *б* — полное неравномерное; *в* — дискоидальное; *г* — поверхностное

Неполное дробление наблюдается в яйцах с большим количеством яиц. При этом типе дробления на бластомеры распадается не все яйцо, а лишь его часть, где содержится ядро, окруженное цитоплазмой — оболочкой новотельной плазмой. Например, в яйцах птиц и рыб образовательная яйцеклетка расположена на одном из полюсов яйца в виде диска. При развитии зародыша этот диск дробится на ряд бластомеров, которые располагаются в один слой, лежащий на массе желтка. В яйцах насекомых образовательная плазма окружает желток, находящийся в центре яйца. Дробление оболочки новотельной плазмы приводит к возникновению бластулы, внутри которой имеется первичная полость — бластоцель. Бластоцель ограничена бластодермами, расположенными в один слой — бластодермой (рис. 7).

Следующий этап развития зародыша заключается в превращении однослоистого зародыша — бластулы — в двухслойный — гаструлу. Этот процесс называется гаструляцией, который у разных животных протекает неодинаково. Но как бы ни шел процесс гаструляции, он

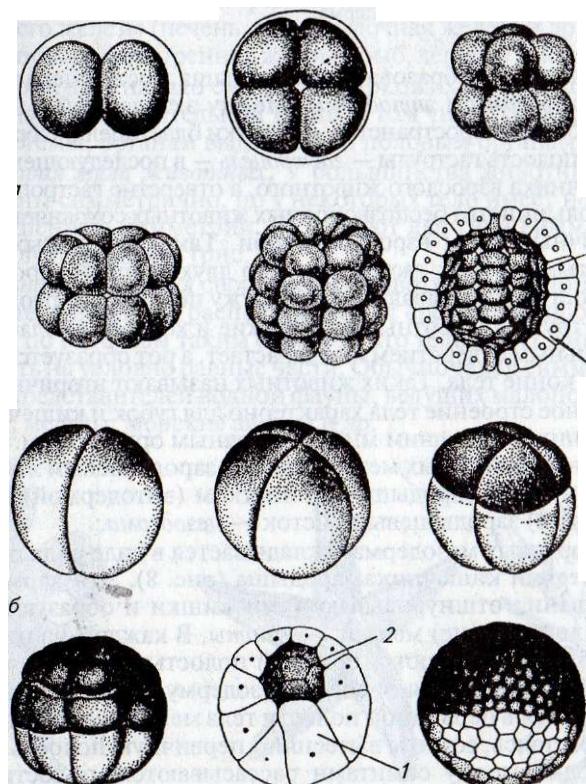


Рис. 7. Схема полного дробления яйца до стадии гаструлы:
а — равномерное дробление яйца голотурии; б — неравномерное дробление яйца лягушки; 1 — бластодерма; 2 — бластоцель

Рис. 8. Продольные и поперечный разрезы через различные эмбриональные стадии ланцетника:

a — бластула; *b, e, g* — гастрula на разных стадиях развития; *d, e, ж* — образование мезодермы, хорды и нервной системы; *f* — анистомальный полюс; *2* — вегетативный полюс; *3* — гастрапльная полость; *4* — гастропор; *5* — первая трубка; *6* — нервно-кишечный канал; *7* — невропор; *8* — складка мезодермы; *9* — целомические мешки; *10* — хорда; *11* — место будущего рта; *12* — место будущего заднего прохода; *13,14* — мезодермальные карманы; *15* — эктодерма; *16* — энтодерма

всегда заканчивается образованием зародыша, состоящего из двух слоев клеток: *эктодермы* и *энтодермы*. Между эктодермой и энтодермой остается небольшое пространство — остатки бластоцеля. Образовавшаяся кишечная полость гаструллы — *гастроцель* — в последующем станет полостью кишечника взрослого животного, а отверстие гастроцеля — *блестопор* — у большинства беспозвоночных животных сохраняется и становится первичным ртом взрослой особи. Таких животных называют первичноротыми. Итак, гастрula — это двухслойный зародыш с кишечной полостью, открывающейся наружу первичным ртом.

У другой группы животных (иглокожие и хордовые) бластопор становится анальным отверстием или застает, а рот образуется на противоположном конце тела. Таких животных называют вторичноротыми.

Двухслойное строение тела характерно для губок и кишечнополостных, относящихся к низшим многоклеточным организмам. У высокоорганизованных животных между первым зародышевым листком (эктодермой) и вторым зародышевым листком (энтодермой) образуется средний (третий) зародышевый листок — *мезодерма*.

У части хордовых мезодерма закладывается в виде ряда парных выпячиваний стенки кишечника зародыша (рис. 8). Эти карманообразные выпячивания отшнуровываются от кишки и образуют мезодермальные (целомические) мешки — *сомиты*. В каждом из них имеется полость, которую называют вторичной полостью тела или *целомом*, а стенки сомитов представляют собой мезодерму. Таким образом, сомиты располагаются в первичной полости тела между эктодермой и энтодермой. Разрастаясь, сомиты вытесняют первичную полость тела, а когда перегородки между сомитами рассасываются, полости сомитов сливаются, образуя вторичную полость тела. Остатки первичной полости сохраняются в виде лакун и каналов. Животные, имеющие целом, называются вторичнopolостными.

Первичнополостные животные не имеют кровеносных сосудов и функцию крови в их организме выполняет полостная жидкость, омывающая внутренние органы.

У примитивных вторичнополостных животных (моллюски, членистоногие) первичная полость тела частично сливается с целомом, образуя смешанную полость тела — миксоцель. Кровеносная система у таких животных незамкнутая, т. е. кровь у них движется то по сосудам, то по лакунам.

В процессе развития эмбриона из зародышевых листков путем дифференцировки образуются ткани и органы. В формировании эпителиальной ткани участвуют все три зародышевых листка. Эктодерма дает начало наружным покровам и их производным (кожные железы, волосы, перья, чешуя, когти), наружному скелету беспозвоночных животных, эпителию переднего и заднего отделов пищеварительной системы, нервной системе и органам чувств, мочевым протокам, висцеральному скелету и наружным жабрам.

Из энтодермы образуются средний (пищеварительный) отдел кишечника и все его железы (печень, поджелудочная железа и др.), хорда, плавательный пузырь и внутренние жабры у рыб, легкие у высших животных.

Мезодерма дает начало скелету у иглокожих и позвоночных животных, мышцам, соединительной ткани, в том числе крови, части кровеносной системы, органам выделения и половым органам.

Симметрия тела животных. У большинства животных части тела расположены симметрично, но у некоторых тело имеет неправильную форму, лишенную симметрии. Различают два типа симметрии: радиальную и двустороннюю (рис. 9).

Радиальная симметрия свойственна животным, у которых одинаковые части тела и органы располагаются от срединной продольной оси животного по радиусам таким образом, что тело таких животных можно разделить на условно равные части. Обычно такая симметрия характерна для представителей водной фауны, ведущих малоподвижный образ жизни: медузы, морские звезды и др.

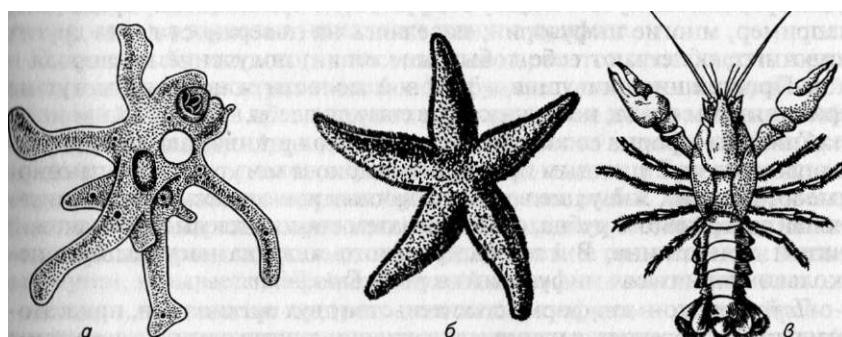


Рис. 9. Типы симметрии:
а — асимметричная (амеба); б — радиальная (морская звезда); в - - двусторонняя (речной рак)

Двусторонняя симметрия характеризуется тем, что тело животного может быть разделено плоскостью только на две равные половины, т. е. у таких животных можно различить левую и правую половины или передний и задний концы тела. Этот тип симметрии всегда бывает относительным, так как в расположении многих внутренних органов ее нельзя достигнуть (сердце у приматов лежит в левой части грудной клетки, а у кур-несушек яичник расположен в левой части таза и т. п.).

Тип симметрии является важным систематическим признаком различных групп животных. Встречаются случаи, когда с возрастом животного меняется симметрия его тела. Например, плавающие личинки морских ежей обладают двусторонней симметрией, а взрослая особь — радиальной.

Симбиоз и паразитизм в животном мире. Отношения животных, обитающих в разных условиях среды, строятся на прямом, непосредственном или косвенном взаимном влиянии. Питание — основной фактор, который определяет взаимоотношения животных. Именно на этой основе складываются пищевые связи между животными и растениями, а также между разными видами животных. По характеру питания одни животные являются плотоядными, другие — растительноядными, третьи — всеядными.

Животные делятся также на *монофагов* и *полифагов*. Монофаги питаются растениями или животными, принадлежащими к одному виду (тля филлоксера питается только соком виноградной лозы). Для полифагов пищей служат растения или животные разных видов (полевка обыкновенная поедает до 100 видов растений). Многие животные используют в пищу существенно ограниченный круг животных или растений. Это так называемые *олигофаги* (например, комары рода анофелес питаются кровью только крупных стадных млекопитающих).

В процессе эволюции органического мира возникли различные формы тесного сожительства разных видов животных, животных и растений. Существуют три формы такого сожительства: комменсаллизм, симбиоз и паразитизм.

Комменсаллизм — такая форма сожительства, при которой только один организм получает пользу от другого, не причиняя ему вреда. Так, например, многие инфузории, поселяясь на поверхности тела других животных, облегчают себе добывание пищи, получение кислорода и т. п. Простейшие, живущие в ротовой полости животных, могут не причинять им вреда, но извлекать пользу для себя.

Симбиоз — форма сожительства, когда оба организма получают взаимную пользу. Типичным примером симбиоза могут служить панцирные инфузории, живущие в рубце жвачных животных: питаясь бактериями содержимого рубца, они сами затем становятся высокобелковой пищей для хозяина. В 1 см³ содержимого желудка насчитывают несколько сотен тысяч инфузорий из рода *Entodinium*.

Паразитизм — это форма сожительства двух организмов, при котором паразит поселяется на поверхности или внутри организма хозяина и питается частями его тела или продуктами пищеварения хозяина. Наружные паразиты называются *экто паразитами*, а паразитирующие

ми у гри организма хозяина — эндопаразитами. Паразитами и их хозяевами могут быть как растения, так и животные. В отличие от хищников паразиты, существуя за счет хозяина, обычно не убивают его, но могут и способствовать его гибели, подрывая его здоровье. Как правило, паразиты по размерам всегда меньше своего хозяина и его организация в равнении с организацией свободноживущих сородичей резко упрощена; особенно это касается нервной системы, органов движения и выхания. Обычно паразит и его хозяин принадлежат не только к разным видам, но даже к разным типам и полцарствам животного мира.

Паразитические животные произошли, вероятно, от свободноживущих предков в процессе постепенного и длительного приспособления к жизни в других организмах. Зачастую предками паразитов были коменсальсы и симбионты, которые постепенно переходили к питанию исключительно за счет хозяев. Огромная масса свободноживущих организмов, которые с пищей попадали в пищеварительный тракт животных, погибли. Но какая-то часть из них, приспособившись к жизни в кишечнике, перешла к паразитическому образу жизни. Из кишечного тракта паразиты могли попадать в кровь, лимфу, различные органы и ткани, становясь постоянными обитателями организма хозяина. Некоторые эктопаразиты, проникав через кожный покров тела хозяина, могли превратиться в илопаразитов. Таким образом, переход свободноживущих форм к паразитическому образу жизни мог быть осуществлен разными путями.

Формы и проявления паразитизма в животном мире разнообразны. Иременный паразитизм характеризуется тем, что паразит вступает в контакт с хозяином только тогда, когда принимает пищу: кровососущие насекомые (комары, слепни, блохи, постельный клоп и др.). В остальное время эти паразиты остаются свободноживущими.

Постоянному паразитизму свойственно то, что паразит проводит на теле хозяина или внутри него всю жизнь на всех стадиях своего развития или только на строго определенной стадии развития. Например, опаснейший паразит круглый червь трихинелла ни на одной стадии развития не покидает своего хозяина — крысу, свинью, человека и др. У оводов же только личинки проходят свое развитие в организме хозяина (лошадь, корова, овца), а взрослые насекомые живут в природе.

Жизненный цикл многих постоянных паразитов протекает не в одном, а в двух и даже в трех хозяевах. Возбудитель малярии малярийный плазмодий обитает в крови человека и в организме комара. Ленточный червь лентец широкий в половозрелом состоянии живет в кишечнике хищных животных и человека, а личиночные стадии его развития проходят в организмах мелких ракообразных и затем рыбы. Таким образом, развитие этого паразита проходит со сменой хозяев. Хозяин, в организме которого паразит достигает половой зрелости и размножается половым путем, называется *дeфинитивным* или *окончательным* хозяином.

Хозяин, в организме которого этот же паразит размножается бесполым путем или проходит стадии своего развития, не достигая половой зрелости, называется *промежуточным*. Следовательно, пораженный широким лентецом человек является дефинитивным хозяином, в ко-

тором происходит половое размножение паразита; мелкие ракообразные — промежуточным, а рыба — дополнительным.

Обитая в организме хозяина, паразитические животные оказывают на него разносторонние и обычно отрицательные воздействия: питаясь за счет организма хозяина, паразит истощает его; выделяемые паразитом продукты обмена вызывают у хозяина токсикозы; механически повреждая ткани и органы, паразиты способствуют проникновению в организм хозяина патогенных микроорганизмов и пр.; паразиты могут разрушать ткани и органы хозяина, закупоривать воздухоносные пути, просвет кишечника, выводные протоки желез; ослабленный паразитами организм хозяина подвержен всевозможным инфекциям.

Паразитарные заболевания, возбудителями которых являются животные организмы, называются *инвазионными* в отличие от *инфекционных* заболеваний, возбудителями которых служат вирусы, бактерии и грибы.

В фауне нашей планеты описано более 80 тыс. видов паразитических животных, из которых половина приходится на перепончатокрылых насекомых — наездников. На втором месте по численности находятся паразитические круглые и плоские черви, и лишь на третьем месте — паразитические простейшие. Среди хордовых паразитов очень мало.

Паразиты локализуются во всех системах органов животных, преобладая в пищеварительной. У растений животные-паразиты, а это в основном нематоды, локализуются также во всех системах органов, но главным образом в корневой системе. Организм хозяина для паразита — это среда его обитания. Внешняя же среда также воздействует на паразита, но через изменения внутренней среды организма хозяина. Удивительную чувствительность к состоянию организма демонстрируют даже эктопаразиты. Паразитическая моногенетическая trematoda многоустка начинает свое развитие на жабрах головастика. Развитие паразита и хозяина — головастика — происходит синхронно. Как только головастик превращается в лягушонка, паразит переходит в мочевой пузырь животного. Если развитие головастика по какой-либо причине задерживается, то замедляется и развитие многоустки.

Распространению паразитов способствуют *резервуарные* хозяева, попадая в организм которых паразиты не развиваются, а накапливаются, сохраняя свою жизнеспособность. Такие резервуарные хозяева известны для многих паразитов из разных классов. Например, установлена роль земляных червей в распространении аскарид: черви заглатывают яйца аскарид и в их кишечнике из яиц выходят личинки, сохраняющие жизнеспособность до года. Съедая такого дождевого червя свиньи или другие животные заражаются аскаридами.

Система животного мира. Животный мир отличается большим разнообразием, на Земле насчитывается около 2 млн видов. Создание системы животного мира является важнейшей задачей зоологии. Решением ее занимается одна из отраслей этой науки — систематика (таксономия), которая разрабатывает теорию и практику классификации и определения животных. Без систематики и ее конечного результата — классификации — все огромное разнообразие видов воспринималось

бы как хаос, недоступный пониманию. Естественная система животного мира строится на основе всестороннего изучения животных, что позволяет выявить не только черты сходства и различия между ними, но и доказать их исторические связи и установить степень родства.

Основной таксономической единицей в систематике является *вид* (*species*) — реально существующая категория. Вид — это обособленная группа сходных особей, обитающих на определенном пространстве (ареале), свободно скрещивающихся между собой и дающих плодовое потомство. Особи разных видов, как правило, между собой не скрещиваются, но если такое скрещивание произойдет, то полученное потомство обычно не способно к дальнейшему размножению.

Каждый вид населяет определенное пространство, называемое областью распространения вида или его ареалом. Особи одного вида, населяющие различные районы ареала, находятся обычно в неодинаковых условиях среды, что приводит к возникновению изменчивости, т. е. приобретению этими особями своеобразных черт. Такие несколько отличные местные группы особей вида, населяющие часть ареала, называются *подвидами*. В отличие от видов подвиды связаны друг с другом переходными формами, обладающими признаками промежуточного характера.

У большинства видов сельскохозяйственных животных выделяют породы, выведенные человеком и отличающиеся, в первую очередь, продуктивностью, но также экsterьерными и интерьерными показателями. Например, во всем мире насчитывают около 400 официально зарегистрированных пород домашних собак.

В современной систематике принято именовать различные виды животных на латинском (или латинизированном греческом) языке, что делает эти названия интернациональными. Впервые двойное (бинарное) название ввел великий шведский ученый К. Линней еще в XVIII в. В соответствии с правилом бинарной номенклатуры каждому виду присваивается название, состоящее из двух слов, первое означает род, второе — собственно вид. Например, различные виды кошек составляют один род *Felis*. Отдельные же виды этого рода будут называться уже двумя словами: например, кот лесной — *Felis silvestris*, кот степной — *F. libysca*, кот камышовый — *F. chaus* и др. После названия вида животного обычно указывается фамилия (полностью или сокращенно) ученого, впервые описавшего данный вид, и год, когда это было сделано. Например, собака домашняя *Canis familiaris* L., 1758., в данном случае L. — это К. Линней.

В современной систематике животных используются следующие таксономические группы (таксоны). Близкие виды объединяются в род (*genus*), близкие роды — в семейство (*familia*), семейства — в отряд (*ordo*), отряды — в класс (*classis*), классы — в тип (*phylum*). Типы образуют царство (*regnum*) животных. Часто устанавливаются промежуточные категории — подрод (между родом и видом), подсемейство (между семейством и родом), подотряд (между отрядом и семейством), подкласс (между классом и отрядом), подтип (между типом и классом). Кроме того, выделяют надсемейства (между семейством и подотрядом), надотряд (между отрядом и подклассом), надкласс (между классом и подтиповом).

Высшая систематическая категория — это тип. Каждый тип характеризуется определенным планом строения, общим для всех групп, входящих в его состав, и общим происхождением.

Тип подразделяется на следующие основные категории: ТИП—подтип—над класс—КЛАСС—подкласс—надотряд—ОТРЯД—подотряд—над семейство—СЕМЕЙСТВО—подсемейство—РОД—подрод—ВИД—подвид.

Царство Животные (Animalia, или Zoa)

Подцарство Одноклеточные, или Простейшие (Protozoa)

- Тип Саркомастигофоры (Sarcomastigophora)
- Тип Апикомплексы (Apicomplexa)
- Тип Миксоспоридии (Mixozoa)
- Тип Микроспоридии (Microspora)
- Тип Асцетоспоридии (Ascetospora)
- Тип Лабиринтулы (Labyrinthomorpha)
- Тип Инфузории (Ciliophora)

Подцарство Многоклеточные (Metazoa)

Надраздел Фагоцителлозои (Phagocytellozoa)

- Тип Пластинчатые (Placozoa)

Надраздел Паразои (Parazoa)

- Тип Губки (Porifera, или Spongia)

Надраздел Эуметазои (Eumetazoa)

Раздел Лучистые (Radiata)

- Тип Кишечнополостные (Coelenterata)
- Тип Гребневики (Ctenophora)
- Тип Мезозой (Mesozoa)

Раздел Двустороннесимметричные (Bilateria)

- Тип Плоские черви (Plathelminthes)
- Тип Круглые черви (Nemathelminthes)
- Тип Немертины (Nemertini)
- Тип Кольчатые черви (Annelida)
- Тип Моллюски (Mollusca)
- Тип Онихофоры (Onychophora)
- Тип Членистоногие (Arthropoda)
- Тип Погонофоры (Pogonophora)
- Тип Щупальцевые (Tentaculata)
- Тип Щетинкочелюстные (Chaetognatha)
- Тип Иглокожие (Echinodermata)
- Тип Полухордовые (Hemichordata)
- Тип Хордовые (Chordata)

Современная систематика выделяет большое число типов животных. В данном учебнике дается описание только тех типов, которые имеют важное значение либо для познания эволюции животных, либо с практической точки зрения.

Царство Животные (Animalia) делят на два полцарства: Простейшие, или Одноклеточные (Protozoa) и Многоклеточные (Metazoa).

ПОДЦАРСТВО ОДНОКЛЕТОЧНЫЕ, ИЛИ ПРОСТЕЙШИЕ (Protozoa)

К одноклеточным относят животных, у которых тело морфологически соответствует одной клетке, но одновременно представляет самостоятельный организм со всеми присущими живому существу функциями. Известно значительное число представителей простейших, образующих колонии из нескольких или многих клеток, но эти Простейшие не могут быть отнесены к многоклеточным организмам, поскольку каждая клетка такой колонии выполняет все функции, хотя в некоторых случаях и намечается разделение отдельных функций между клетками колонии. В многоклеточном же организме каждая клетка выполняет определенную функцию — двигательную, нервную и т. п.

Деление клеток у многоклеточных животных приводит к росту организма, а деление клетки у простейших приводит к увеличению их численности, т. е. деление у простейших — это по сути дела размножение этих организмов.

Известно более 39 тыс. видов одноклеточных. Это мелкие организмы. Минимальными размерами характеризуются простейшие, ведущие паразитический образ жизни внутри клеток растений и животных (эндопаразиты) — всего 2—4 мкм. Тело простейших ограничено снаружи тончайшей мембраной (или более плотной и эластичной пелликулой), под которой находятся цитоплазма и ядро (одно или несколько). Цитоплазма представлена двумя слоями: наружным светлым и плотным — эктоплазмой, и внутренним менее плотным с многочисленными включениями — эндоплазмой. В эндоплазме сосредоточены все основные органеллы клетки: митохондрии, рибосомы, лизосомы, аппарат Гольджи, эндоплазматическая сеть и пр.

У простейших имеются специальные органеллы: пищеварительные и сократительные вакуоли, опорные и сократительные фибриллы.

Простейшие, тело которых ограничено мембраной, не имеют постоянной формы (амебы). У ряда видов клеточная мембра уплотняется за счет эктоплазмы и становится плотной и эластичной, образуя так называемую пелликулу. В этом случае животные имеют определенную форму тела (инфузории) и одновременно сохраняют достаточную гибкость. Часть одноклеточных имеет постоянную форму тела благодаря укреплению оболочки за счет различных включений.

Функцию скелета у простейших могут выполнять раковины, формирующие наружный скелет, или специальные иглы и капсулы, формирующие внутренний скелет. Раковины образуются из веществ, выделяемых эктоплазмой, а внутренний скелет возникает в эндоплазме клетки. Основу скелетных образований составляют органические и минеральные вещества (CaCO_3 , SiO_2 , SrSO_4).

Самый простой способ движения простейших, не имеющих постоянной формы тела, — движение с помощью ложноножек, или псевдоподий (амебоидное движение). Псевдоподии — это выросты клетки, в которые перетекает цитоплазма. Более сложное движение характерно для простейших, обладающих жгутиками или ресничками. Строение жгутиков и ресничек сходно, но для жгутиков свойственно вращательное движение, а для ресничек — гребной тип движения. Внутриклеточные паразиты, как правило, не имеют органелл, обеспечивающих движение.

Разнообразен тип питания простейших. Среди них встречаются автотрофы, которые способны к фотосинтезу (одноклеточные жгутиконосцы). Но большая часть простейших — гетеротрофы, питающиеся готовыми органическими веществами. Одним простейшим свойствен голозойный способ питания путем проглатывания оформленных частиц пищи, другим — сапрофитный способ за счет поглощения растворенных органических соединений.

Когда в клетку простейшего поступают оформленные пищевые частицы, вокруг них образуются пищеварительные вакуоли, в которых эти частицы перевариваются. Такой захват частиц клеткой получил название фагоцитоза. При сапрофитном способе питания пищеварительных вакуолей в организме простейших не образуется. Захват клеточной поверхностью растворенных органических веществ называется пиноцитозом.

Небольшое число простейших обладает смешанным (миксотрофным) типом питания. В одних условиях они способны к фотосинтезу, в других — к питанию органическими веществами, т. е., имея в цитоплазме хлорофилловые зерна, они могут образовывать и пищеварительные вакуоли.

У пресноводных простейших процессы осморегуляции и выделения осуществляются с помощью сократительных вакуолей. У паразитических и морских форм сократительные вакуоли отсутствуют, так как среда, в которой обитают эти животные, и их внутреннее содержимое изотоничны. Выделение продуктов обмена у большинства простейших происходит через поверхность клетки, а также через сократительные вакуоли. Кислород поступает в клетку путем диффузии через клеточную мембрану.

В цитоплазме большинства простейших находится одно, реже два или несколько ядер, которые регулируют обмен веществ и размножение. Ядра одноклеточных имеют те же структуры и компоненты, что и ядра клеток многоклеточных животных, хотя и характеризуются морфологическим многообразием. У части многоядерных простейших ядра выполняют различные функции: репродуктивные и вегетативные. Такое явление называют ядерным дуализмом (у инфузорий). При бесполом размножении деление ядра у одноклеточных происходит по типу митоза

(непрямого деления), позволяющего сохранять преемственность в ряду Клеточных поколений. Ядра простейших, которым свойствен половой **процесс**, делятся путем мейоза, или редукционного деления.

Размножаются простейшие бесполым и половым путями. Бесполое размножение происходит путем деления клетки на две или множество **клеток**. Половой процесс заключается в образовании половых клеток — **гамет** (женских — макрогамет и мужских — микрогамет) и их слияния. В результате образовавшаяся зигота дает начало новому дочернему организму. У инфузорий половой процесс происходит путем коньюгации — слияния генеративных ядер двух особей, а не половых клеток.

Жизнь одноклеточных животных представлена рядом стадий, которые чередуются с определенной закономерностью. Период жизни организма между двумя одинаковыми стадиями называется жизненным циклом данного вида. Обычно жизненный цикл начинается со стадии зиготы (соответствует оплодотворенной яйцеклетке многоклеточных Животных), затем следуют стадия бесполого размножения делением, стадия образования половых клеток-гамет, которые, сливаясь попарно, дают новую зиготу. При бесполом размножении жизненный цикл — это период от деления до деления, только при половом размножении **жизненный цикл** — это период от зиготы до зиготы.

Важнейшая биологическая особенность одноклеточных — образование цист, или инцистирование. Для сохранения жизнеспособности в неблагоприятных для организма условиях животные округляются, теснясь воду, образуют плотную оболочку и переходят в состояние покоя, в таком состоянии простейшие могут длительный период сохранять жизнеспособность, пассивно перемещаться на большие расстояния с воздушными массами, водой и т. п., а в благоприятных условиях вновь перейти к активному образу жизни.

Одноклеточные животные приспособлены к обитанию в разнообразных средах, но для этих животных, как правило, необходимо наличие воды: морские и пресные водоемы, влажные почвы; есть одноклеточные, перешедшие к паразитическому образу жизни в растениях, животных и человеке.

Подцарство Protozoa делят на семь типов, из которых наибольший интерес представляют следующие: Саркомастигофоры (*Sarcomastigophora*), Апикомплексы (*Apicomplexa*), Миксоспоридии (*Micrastrea*), Микроспоридии (*Microspora*) и Инфузории (*Ciliophora*).

ТИП САРКОМАСТИГОФОРЫ (*Sarcomastigophora*)

К саркомастигофорам относят свободноживущих или паразитических одноклеточных животных, которые передвигаются с помощью особых временных выростов цитоплазмы (псевдоподий) или бичевидных выростов (жгутиков). Некоторые одноклеточные могут перемещаться как с помощью псевдоподий, так и с помощью жгутиков. Большинству

видов одноклеточных свойственно только бесполое размножение. Для некоторых простейших свойствен половой процесс путем копуляции.

Тип *Sarcomastigophora* представлен двумя подтипами: Жгутиконосцы (*Mastigophora*) и Саркодовые (*Sarcodina*).

Жгутиконосцы, видимо, стоят ближе к предковым группам простейших. Они разнообразнее по типам питания, органелл движения, типам оболочек клеток и т. п. О первичности жгутиковых форм свидетельствует и то, что саркодовые, которые размножаются половым путем, проходят жгутиковую стадию гамет. Среди жгутиконосцев есть переходные формы между одноклеточными растительными и животными организмами.

ПОДТИП ЖГУТИКОНОСЦЫ (*Mastigophora*)

Жгутиконосцы обитают в морских и пресных водах, в почве и в организме растений и животных; среди них есть опасные паразиты животных и человека. Насчитывают более 8 тыс. видов жгутиконосцев. Растительные и животные жгутиконосцы являются важным звеном в пищевых цепях водных экосистем. Некоторые жгутиконосцы находятся в симбиотических отношениях с различными животными.

Представители жгутиконосцев характеризуются наличием особых органелл — жгутиков, которые служат для передвижения; число жгутиков колеблется от 1,2,4,8 до нескольких тысяч. При этом жгутики имеются постоянно в течение большей части жизненного цикла. У некоторых видов жгутиконосцев помимо жгутиков могут иметься временные или постоянные ложноножки — псевдоподии; этот признак сближает их с представителями подтипа Саркодовые.

Размеры и форма тела жгутиконосцев разнообразны. Они покрыты довольно плотной и сложной по строению оболочкой — пелликулой, что позволяет им сохранять более или менее постоянную форму. Цитоплазма делится на два слоя: эктоплазму и эндоплазму. У видов, способных образовывать псевдоподии, тело покрыто тонкой и эластичной мембраной. У растительных жгутиконосцев оболочка может состоять из клетчатки; у некоторых видов образуется панцирь разнообразной формы, нередко несущий отростки.

От переднего полюса тела отходят жгутики; если жгутиков много (несколько тысяч), то они могут покрывать все тело простейшего. У некоторых жгутиконосцев жгутик тянется вдоль тела, соединяясь с ним с помощью тонкой цитоплазматической мембранны — ундулирующей мембранны. Эта мембрана обеспечивает поступательное движение простейшего в вязкой среде, которое как бы ввинчивается в эту среду. Нижняя часть жгутика, погруженная в эктоплазму, называется базальным тельцем или кинетосомой.

Размножаются жгутиконосцы путем продольного деления клетки на две дочерние; у некоторых представителей существует половой процесс с образованием гамет и последующей их копуляцией.

Среди жгутиконосцев есть автотрофы, которые способны к фотосинтезу, гетеротрофы, характеризующиеся животным типом питания, И, наконец, миксотрофы, сочетающие растительный и животный способы питания. Гетеротрофным жгутиконосцам свойственно либо голозойное (анимальное) питание путем заглатывания частиц органической пищи, либо сапрофитное — за счет всасывания жидкой органической пищи всей поверхностью тела.

Жгутики служат не только для движения, но и помогают захватывать пищевые частицы. В результате движения жгутика в воде возникает водоворот, увлекающий мелкие пищевые частицы к основанию жгутика, где у некоторых видов находится клеточный рот, ведущий в глотку. У видов, не имеющих клеточного рта, у основания жгутика есть участок липкой цитоплазмы, не покрытый пелликулой, через которую пища попадает в организм одноклеточного. Поступившая в цитоплазму пища заключается в образовавшиеся пищеварительные вакуоли. Непереваренные остатки пищи выбрасываются из тела простейшего во внешнюю среду в любом участке клетки.

КЛАСС РАСТИТЕЛЬНЫЕ ЖГУТИКОНОСЦЫ (*Phytomastigophorea*)

Для представителей этого класса характерен автотрофный или миксотрофный типы питания, реже среди растительных жгутиконосцев встречаются виды с гетеротрофным типом питания. Обитают эти жгутиконосцы в соленой и пресной воде. У них имеются хроматофоры, содержащие хлорофилл, а многие имеют светочувствительный глазок — стигму, позволяющий выбирать наиболее освещенные участки водоема для оптимизации процессов фотосинтеза. У пресноводных форм имеется сократительная вакуоль. Встречаются колониальные формы. Колонии образуются при неполном делении, в результате которого не полностью отделившиеся друг от друга особи остаются связанными друг с другом. Колонии могут различаться по форме (шаровидные, древовидные) и по характеру развития (монотомические и палинтомические колонии).

При монотомическом развитии после бесполого размножения путем деления дочерние клетки растут и снова периодически делятся, тем самым увеличивая число особей в колонии, которая в свою очередь периодически делится пополам. При палинтомическом развитии все клетки колонии или только их часть последовательно делятся, но без стадий роста и увеличения их объема, в результате чего образуется сразу несколько молодых колоний. Материнская колония затем распадается на дочерние, число которых соответствует числу клеток старой материнской колонии.

Половой процесс распространен преимущественно у растительных форм, имеющих палинтомический тип колоний. Наиболее сходен с половым процессом многоклеточных половой процесс у колоний *Volvox* (рис. 10). В колонии только немногие клетки дают начало мужским (микрогаметы) и женским (макрогаметы) гаметам. Среди вольвоксовых встречаются раздельнополые виды, в колонии которых обра-

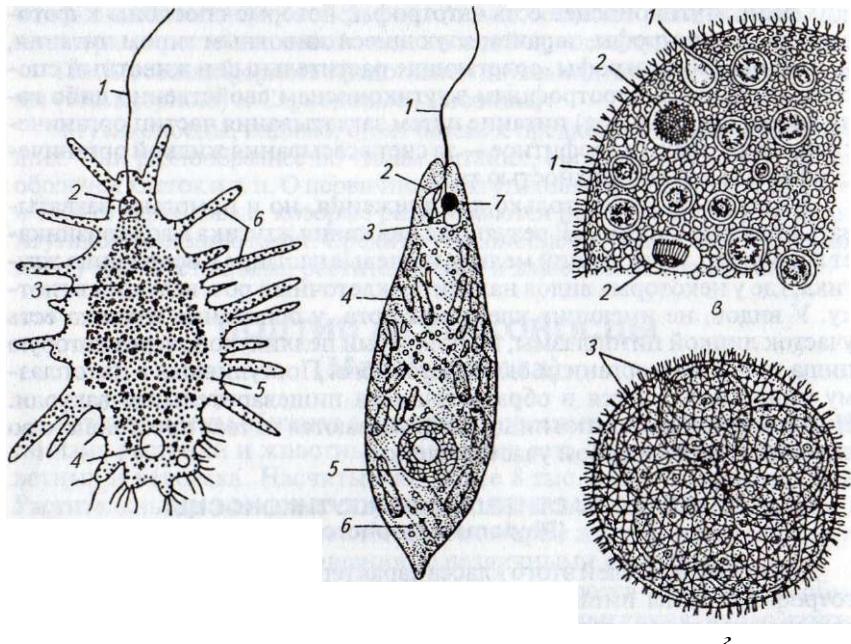


Рис. 10. Растительные жгутиконосцы:
 а — жгутиконосец *Mastigamoeba aspera* (по Шульце); 1 — жгутик; 2 — ядро; 3 — эндоплазма; 4 — сократительная вакуоль; 5 — эктоплазма; 6 — псевдоподии; б — эвглена зеленая (*Euglena viridis*) (по Дофлейну); 1 — жгутик; 2 — резервуар сократительной вакуоли; 3 — сократительная вакуоль; 4 — хроматофоры; 5 — ядро; 6 — зерна парамиля; 7 — глазок;
 в — вольвокс (*Volvox globator*), участок колонии с половыми клетками (по Кону); 1 — макрогаметы; 2 — микrogаметы; 3 — дочерние колонии

зуются или только мужские, или только женские гаметы, и герmafродитные виды, в колонии которых одновременно развиваются и мужские, и женские гаметы. Макрогаметы неподвижны. Микрогаметы отыскивают их и сливаются с ними. Оплодотворенная макрогамета (зигота) дает начало новой колонии путем последовательных плинтомических делений.

К растительным жгутиконосцам относятся: хризомонады, панцирные жгутиконосцы, эвгленовые и вольвоксовые (рис. 10). Растительные свободноживущие морские и пресноводные жгутиконосцы входят в состав планктона, среди них имеются симбионты коралловых полипов и паразитические виды. Пресноводные окрашенные жгутиконосцы при массовом размножении вместе с микроскопическими водорослями могут вызывать «цветение» воды в пресных водоемах, вследствие чего иногда возникают ночные заморы рыбы.

КЛАСС ЖИВОТНЫЕ ЖГУТИКОНОСЦЫ (Zoomastigophorea)

Всем животным жгутиконосцам свойствен гетеротрофный тип питания. Большая их часть являются паразитами растений и животных. Особенно опасны эндопаразиты животных и человека, относящиеся к отряду Кинетопластиды (Kinetoplastida). В плазме крови животных и человека паразитируют различные виды трипаносом (*Trypanosoma*), имеющих лентовидное тело с одним (реже с двумя) жгутиком (рис. 11). В эктоплазме жгутик связан с кинетопластом, а снаружи свободным концом направлен вдоль тела жгутиконосца, срастаясь с ним с помощью цитоплазматической ундулирующей мембранны. На переднем конце жгутик остается свободным. У ряда форм трипаносомных жгутиконосцев жгутик может отходить от середины клетки, у некоторых он начинается на переднем конце тела, а у лейшманий жгутик отсутствует.

Трипаносомы паразитируют в основном в крови и спинномозговой жидкости животных и человека, вызывая тяжелейшие заболевания, называемые трипаносомозами. В тропической Африке трипаносомы *Trypanosoma rhodesiense* и *T. brucei gambiense* вызывают «сонную болезнь» человека. Это длительное и тяжелое заболевание, унесшее свыше миллиона жизней, начинается небольшой лихорадкой, сопровождается сонливостью и постепенно приводит к полному истощению организма человека. Без лечения болезнь заканчивается смертью.

Переносчиками возбудителей сонной болезни являются кровососущие мухи цеце (*Glossina palpalis* и *Gl. morsitans*). Вместе с кровью больного муха засасывает трипаносом, которые размножаются в ее кишечнике и [акапливаются в слюнных железах и хоботке насекомого, являющемся одновременно переносчиком и вторым хозяином паразита. При укусе мухой цеце здорового человека трипаносомы вместе со слюной попадают в его кровь. Природным резервуаром трипаносом являются антилопы и другие животные, почти не страдающие от этих жгутиконосцев, но являющиеся их носителями.

В Южной Америке трипаносома *T. cruzi* вызывает у людей болезнь Чагаса. Переносчиком и вторым хозяином возбудителя являются кровососущие клопы. Трипаносомы выделяются с экскрементами клопа и, попадая в ранку на коже человека, могут заразить его. Паразиты живут в крови, а затем проникают в клетки внутренних органов, где размножаются и снова попадают в кровь.

Есть немало видов трипаносом, вызывающих тяжелые заболевания у крупного рогатого скота и верблюдов. В Африке *T. brucei* поражает рогатый скот, вызывая болезнь Нагана. Переносчиком возбудителя

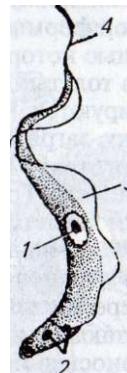


Рис. 11. Трипаносома:
1 ~ ядро; 2 ~ базальный диск;
3 ~ ундулирующая мембра;
4 ~ жгутик

являются мухи цеце. В Южной Азии и Африке кровососущие слепни переносят трипаносому *T. evansi* — возбудителя болезни верблюдов. *T. equiperdum* вызывает случную болезнь лошадей, заражение которых происходит при случке. Паразиты поражают нервную систему животных. Случная болезнь встречается в Средиземноморье, Азии и Африке.

Среди представителей Kinetoplastida есть родичи трипаносом — лейшмании (*Leishmania*), вызывающие лейшманиоз у человека. Это внутриклеточные паразиты с редуцированным жгутиковым аппаратом. Переносчиком лейшманий являются кровососущие москиты, в кишечнике которых паразит размножается и проходит стадию со жгутиком, но без ундулирующей мембранны. При укусе москитами в кровь человека попадают лейшмании, имеющие жгутики. После внедрения в клетки печени и селезенки паразиты утрачивают жгутик.

Один вид лейшманий — *L. donovani* — вызывает у человека заболевание, называемое висцеральным лейшманиозом (кала-азар). Распространенное в Средней Азии, Иране, Южной Америке и Индокитае это заболевание поражает в основном детей, у которых увеличиваются в размерах печень и селезенка. Болезнь сопровождается лихорадкой, истощением и малокровием. Природным резервуаром паразита являются в основном бродячие собаки.

В средней Азии и Закавказье *L. tropica* вызывает восточную язву, или пендинку. Переносчиком и вторым хозяином являются москиты рода *Phlebotomus*, в желудке которых лейшмании размножаются, образуя жгутиконосную форму. В местах укусов москитами образуются изъязвления. Внутри лейкоцитов, находящихся в язве, обнаруживается множество паразитов, лишенных жгутика. Через 1—2 года язвы зарубцовываются. Носителем кожного лейшманиоза могут быть различные грызуны, чаще всего большие песчанки. Висцеральный и кожный лейшманиозы дают стойкий иммунитет.

В кишечнике и желчных протоках человека паразитирует лямблия *Giardia intestinalis*, вызывая болезнь лямблиоз. Тело этого паразита грушевидной формы, имеет несколько жгутиков и вооружено присоской, с помощью которой лямблия прикрепляется к слизистой кишечника. Попав в толстый отдел кишечника, лямблии отбрасывают жгутики и инфицируются. Человек (чаще всего дети) заражается, потребляя пищу и воду, загрязненные цистами паразитов.

Среди отряда Трихомонадовые (Trichomonadida) есть опасные паразиты человека. Трихомонады имеют четыре—шесть жгутиков, из которых один является рулевым и образующим ундулирующую мембрану. *Trichomonas hominis* вызывает хронические поносы, *T. vaginalis* паразитирует в мочеполовых путях, вызывая трудноизлечимые заболевания.

Интересны представители отряда Многожгутиковые (Hypotermastigida), обитающие в кишечнике насекомых, в частности термитов. У этих жгутиконосцев много жгутиков, и они являются полезными симбионтами термитов. Без многожгутиковых хозяева не могут самостоятельно переваривать клетчатку, так как жгутиконосцы вырабатывают фермент целлюлазу, расщепляющую потребленную термитами клетчатку до легко усвояемых углеводов.

ПОДТИП САРКОДОВЫЕ (*Sarcodina*)

Представители саркодовых на протяжении жизненного цикла или большей его части передвигаются с помощью псевдоподий; жгутиками саркодовые могут обладать лишь на кратковременных стадиях развития, это гаметы и зооспоры. Основная масса саркодовых размножается бесполым путем: делением на две клетки или на множество клеток. Половое размножение присуще немногим видам и осуществляется путем слияния гамет. Большинство саркодовых — свободноживущие виды, обитающие в соленых водах, часть из них живет в пресных водоемах, застеляет почву, участвуя в почвообразовательных процессах, и немногие являются паразитами животных и человека. Всего насчитывают около 10 тыс. видов саркодовых.

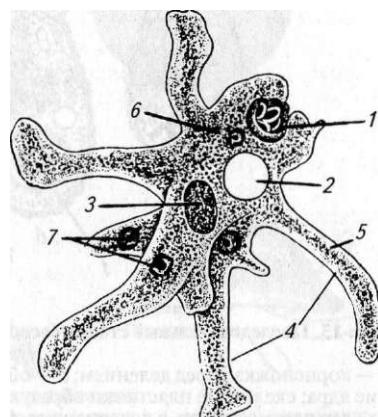
Из 12 классов саркодовых можно выделить три класса, к которым принадлежит основная масса представителей этого подтипа: класс Корненожки (*Rhizopoda*), класс Лучевики (*Radiolaria*) и класс Солнечники (*Heliozoa*).

Наиболее просто устроены представители голых амеб (класс Корненожки — *Rhizopoda*, отр. Амебы — *Amoebina*), населяющие пресные водоемы и почвы, где они питаются мелкими простейшими, одноклеточными водорослями, микроорганизмами и гниющими остатками. Тончайшая мембрана, покрывающая тело этих микроскопических животных, позволяет им образовывать временные выросты — псевдоподии, по своей форме напоминающие корни растений, что и определило название класса. С помощью псевдоподий амебы медленно перетекают с одного места на другое. При этом они обтекают мелкие пищевые частицы (одноклеточные водоросли, бактерии и др.) со всех сторон, и те оказываются внутри цитоплазмы амебы, где образуются пищеварительные вакуоли. С помощью пищеварительных ферментов пищевые частицы перевариваются (внутриклеточное пищеварение). Жидкие продукты переваривания поступают в эндоплазму, а непереваренные остатки транспортируются к поверхности тела и выбрасываются наружу через мембрану клетки. Типичным представителем отряда является пресноводная амeba (*Amoeba proteus*) (рис. 12). Подобный способ захвата

Рис. 12. Амеба (*Amoebaproteus*), захватывающая пищу:

1 — захваченная псевдоподиями пищевая частица; 2 — сократительная вакуоль; 3 — ядро; 4 — псевдоподии; 5 — эктоплазма; 6 — эндоплазма; 7 — пищеварительные вакуоли

3 - 6407



33

пищевых частиц с помощью псевдоподий называют фагоцитозом. Наряду с ним существует способ поступления жидких веществ в тело амебы — пиноцитоз. При этом внутрь цитоплазмы впячивается тонкий канал, в который засасывается капелька жидкости с растворенными в ней органическими веществами. Образовавшаяся вокруг этой капельки вакуоль с жидкостью отшнуровывается от канала и после всасывания жидкости эта вакуоль прекращает свое существование.

Для поддержания осмотического давления у амеб, особенно обитающих в пресных водах и почве, есть особый аппарат для удаления излишков воды из организма — сократительная вакуоль; обычно сократительная вакуоль бывает одна, реже две. У морских и паразитических амеб сократительные вакуоли отсутствуют или пульсируют очень редко. Помимо регулирования осмотического давления сократительные вакуоли участвуют в процессах выделения продуктов обмена и дыхания, обеспечивая организм кислородом из окружающей воды.

Амебам присущее бесполое размножение путем деления на две клетки или на несколько дочерних особей (рис. 13).

При неблагоприятных условиях амебы инфицируются, выделяя вокруг тела плотную оболочку. При наступлении благоприятных условий среды цисты разрушаются и амебы начинают вести активный образ жизни.

В кишечнике позвоночных животных, в том числе домашних животных и человека, обитает множество видов амеб, не принося вреда своим хозяевам, а просто являясь их квартирантами. Большая их часть питается содержимым кишечника, в том числе бактериями (*Entamoeba coli*). Среди

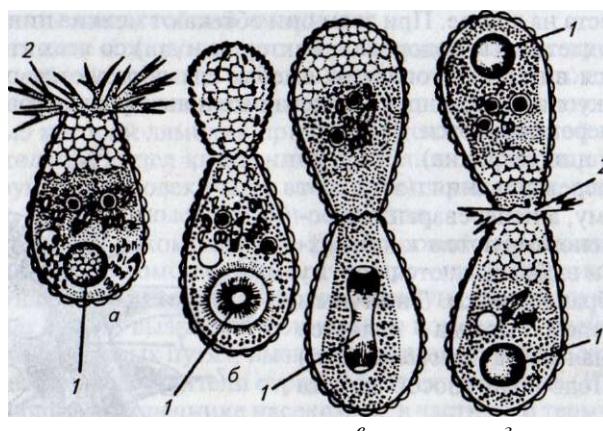


Рис. 13. Последовательные стадии бесполого размножения пресноводной корненожки *Euglypha alveolata*:
а — корненожка перед делением; б — образование цитоплазматической почки; в — деление ядра; скелетные пластинки образуют новую раковину; г — окончание деления; одно из ядер переместилось в дочернюю особь: 1 — ядро; 2 — псевдоподии

таких в толстом кишечнике человека, есть виды, которые могут вызвать тяжелые заболевания, например дизентерийная амеба *E. histolytica* (хронично пищей ей служат бактерии, но в ряде случаев эта амеба может паразитировать под слизистую оболочку кишечника, где питается и размножается, вызывая кровавый понос (кишечный амебиаз). Симптомы заболевания схожи с симптомами дизентерии, поэтому эту амебу называют дизентерийной. С каловыми массами наружу выходит множество цист, которые в наши времена остаются инвазионными (сохраняют способность к заражению). Некоторые люди могут быть носителями дизентерийных амеб.

У представителей отряда Раковинные амебы (Testacea) тело заключено в раковину, образованную органическими рогоподобными веществами, немыми цитоплазмой; зачастую в такую раковину включены песчинки и другие посторонние частицы. Раковины имеют отверстие — устье, из которого амебы выдвигают псевдоподии. Раковинные и голые амебы в Олипопом количестве населяют пресные водоемы, сфагновые мхи и почву, участвуя в процессах почвообразования. Благодаря своим микроскопическим размерам они способны существовать в тончайшем водном слое, окружающем частички почвы. При пересыхании почвы амебы инцистируются и в виде цист могут переноситься ветром с пылью на значительные расстояния. В благоприятных условиях почвенные амебы быстро размножаются и делением надвое: одна из клеток остается в материнской раковине, а другая строит себе новую раковину. У увлажненных и заболоченных почв наиболее многочисленны арцелла и диффлюгия (рис. 14).

Более сложно устроены обитатели морей из отряда Фораминиферы (Foraminifera), у которых раковина образуется из веществ (близких по

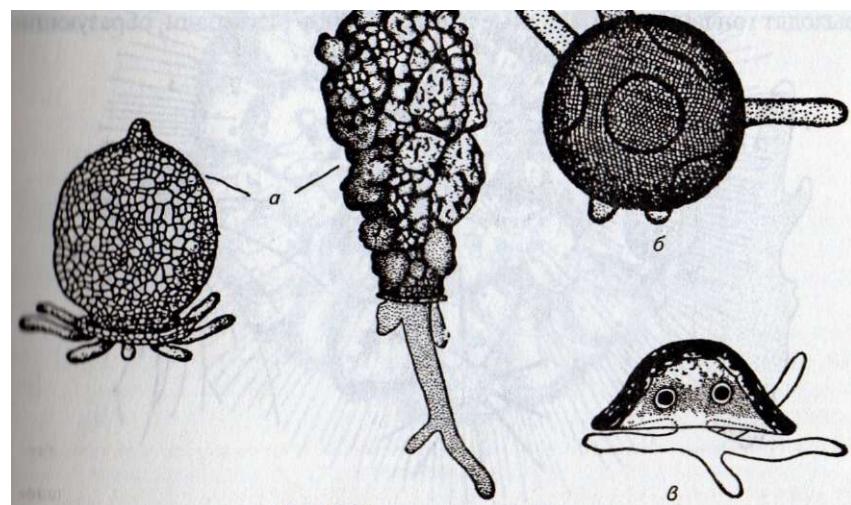


Рис. 14. Раковинные корненожки:
а — диффлюгия (*Diffugia*); б — арцелла (*Arcella*), вид сверху; в — арцелла, вид сбоку

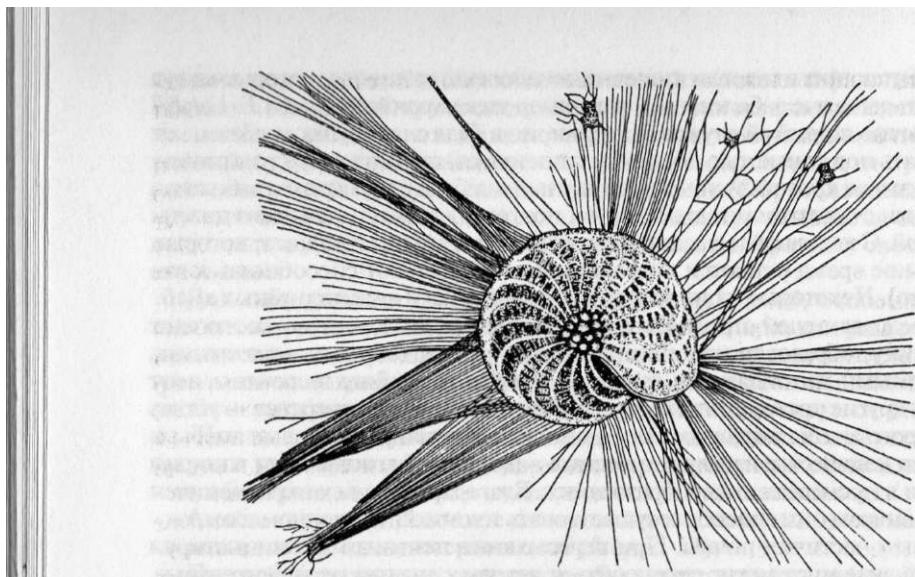


Рис. 15. Форамиинифера. Раковина и сеть тончайших псевдоподий

природе к псевдохитину), выделяемых эктоплазмой. Это самая многочисленная и разнообразная группа саркодовых, встречающихся во всех морях на значительной глубине. У некоторых видов псевдохитиновые раковины инкрустированы песчинками, у других пропитаны углекислым кальцием (рис. 15). Форма раковин разнообразна, внутри они имеют одну или несколько сообщающихся камер, в которых находится тело корненожки. Кроме устья у раковины имеется множество пор, через которые наружу выходят тончайшие нитевидные псевдоподии — изоподии, образующие

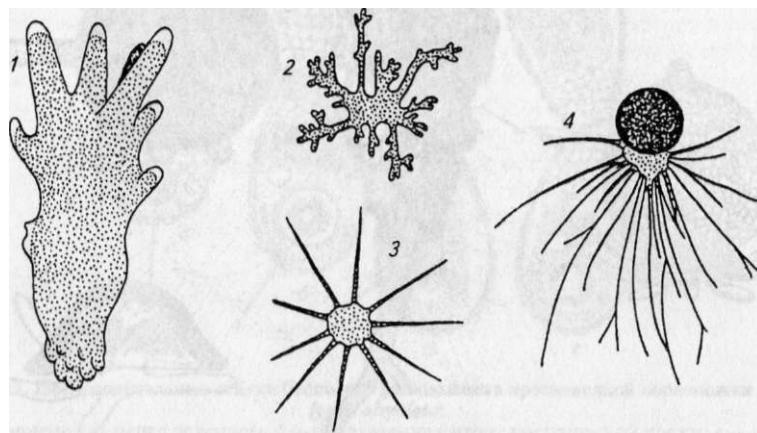


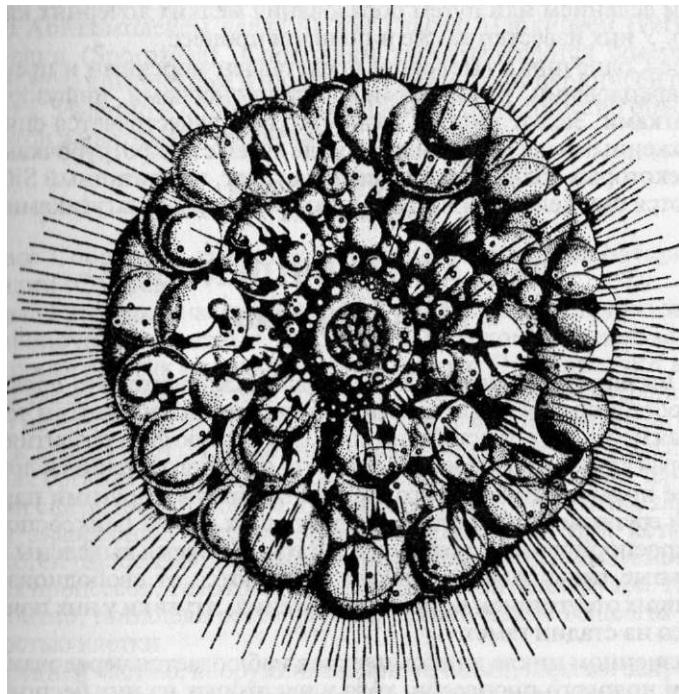
Рис. 16. Типы псевдоподий у саркодовых:
1 — лобоподии; 2 — ризоподии; 3 — аксоподии; 4 — филоподии

• и it ꙗї раковины сложную цитоплазматическую сеть; с помощью этой се-
м I прмсможки передвигаются и питаются (рис. 16). Пищей им служат
п,д ігрті, мелкие простейшие и даже некоторые многоклеточные.

II приду с бесполым размножением фораминиферы размножаются и
нши.иным путем. Сначала тело корненожки распадается на множество
нм>1н1||д||ых клеток, которые покидают материнскую раковину, растут
и пїаю вокруг себя новую раковину. Дочерние корненожки дают
иічнико /ругому поколению раковинных корненожек: путем множест-
и о деления образуют гаметы — мелкие клетки с двумя жгутиками.
'Ігрт I поры раковины одинаковые по форме и размерам гаметы выхо-
14 1 н иоду и попарно сливаются, образуя зиготу, которая дает начало
•____v поколению. Таким образом, в жизненном цикле фораминифер
Происходит чередование бесполого и полового размножения.

1.1нышая часть фораминифер живет в придонном слое морей и Океа-
нии, входя в состав бентоса и питаясь мелкими организмами. Немногие
шнин!, обладающие легкой раковиной, входят в состав планктона.

II иерхних слоях морей живут саркодовые со сложным внутренним
• и¹ не том (класс Радиолярии, или Лучевики — Radiolaria). Болынинст-
1н и I них имеет окружной формы тело, от которого в виде лучей отходят
ммочисленные тонкие псевдоподии (рис. 17). Часто у радиолярий в



І'm 17. Радиолярия *Thalassophysa pelagica*. В центре видны крупное ядро и центральная капсула

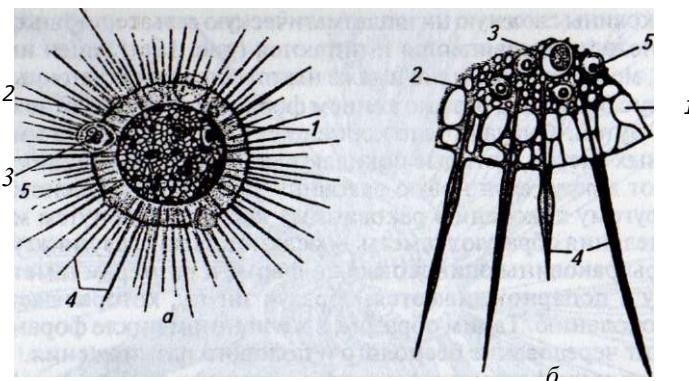


Рис. 18. Солнечник *Actinosphaerium eichhorni*:
а — общий вид; б — участок тела при большом увеличении; 1 — эктоплазма; 2 — эндоплазма; 3 — пищевой комок; 4 — аксоподии; 5 — ядро

цитоплазме находятся симбионты — водоросли, снабжающие хозяина кислородом и сами служащие ему пищей. Радиолярии размножаются простым делением или путем образования мелких дочерних клеток зооспор. У них известен также и половой процесс.

Класс Солнечники (Heliozoa) представлен морскими и пресноводными саркодовыми, питающимися жгутиконосцами, инфузориями и коловратками. Захват пищи у солнечников осуществляется спирально расположенным псевдоподиями в виде лучей с микротрубочками (рис. 18). У некоторых солнечников имеется скелет, пропитанный SiO_2 . Размножаются они делением или образуют зооспоры со жгутиками.

ТИП АПИКОМПЛЕКСЫ (Apicomplexa)

Тип насчитывает около 4,8 тыс. видов исключительно паразитических простейших, среди которых много паразитов животных и человека. Основная масса этих простейших проходит особую фазу развития — спору, которая служит для перехода паразита от одного хозяина к другому.

Ранее представителей этого типа объединяли с другими паразитическими группами простейших, образующих споры (миксоспоридиями, микроспоридиями), которые в настоящее время выделены в самостоятельные типы. Апикомплексы отличаются от свободноживущих простейших отсутствием органелл движения, жгутики у них появляются только на стадии гамет.

В жизненном цикле апикомплексов наблюдается чередование бесполого и полового процессов, хотя у некоторых из них бесполое размножение отсутствует. Бесполое размножение осуществляется путем

Множественного деления — шизогонии, в результате чего образуются **меротиты**. Мерозоиты инфицируют здоровые клетки хозяина. В **последующем** новые поколения мерозоитов дают начало поколению **половых особей** — гамонтов, формирующих половые гаметы.

Половой процесс протекает в форме копуляции гамет, которые у Оомыиинства апикомплексов различаются по размерам, т. е. образуются **Микро-** и микрогаметы. Зигота одевается плотной оболочкой и называется **Нут** ооцистой. В ооцисте начинается процесс спорогонии — образования множества спорозоитов, которые находятся внутри спор, покрытых **ОЛственной** оболочкой. Образованием спорозоитов заканчивается ШИ ннший цикл апикомплексов.

Таким образом, шизогония ведет к увеличению числа паразитов в тканях хозяина, а спорогония способствует росту числа паразитов в период их расселения в вдвое ооцист со спорами. Ооцисты и споры покрыты плотными оболочками, защищающими спорозоиты от внешней среды.

И жизненном цикле части споровиков происходит смена хозяев. Метацисты и спорозоиты для проникновения в клетки хозяина имеют особый апикальный комплекс органелл на переднем конце тела (отсюда и название типа — Апикомплексы), представляющий собой упругую ОНИриль, которая проникает в клетку хозяина после растворения оболочки этой клетки особым секретом, синтезируемым паразитом.

Тип Апикомплексы делят на два класса: Перкинсеи (Perkinsea) и Споровики (Sporozoea). Споровики характеризуются совершенным **Апикальным** комплексом и наличием в отличие от перкинсеи полового процесса. Именно к споровикам относится большинство опаснейших **Паразитов** животных и человека.

КЛАСС СПОРОВИКИ (*Sporozoea*)

Класс Споровики включает два отряда: Грегариниды (Gregarinida) и Коццидии (Coccidia).

О град Грегарини насчитывает более 500 видов — паразитов многих позвоночных животных, в основном насекомых и кольчатых червей, а также водных моллюсков и иглокожих. Паразитируют в кишечнике, а также в полости тела и гонадах. Тело кишечных грегарин разд蹭ено на три участка: передний, средний и задний (эпимерит, протомерит и дейтомерит). Грегарини, паразитирующие в полости тела и в поповых органах, не обладают трехчленностью, их тело червеобразной или сферической формы. Грегарини — эндопаразиты, характеризующиеся анаэробным (бескислородным) дыханием, при котором паразитикоген расщепляется с выделением энергии, используемой для обменных процессов. Тело грегарин одето плотной пелликулой. Питаются оипрофитно, поглощая растворенные органические вещества всей поверхностью клетки.

Передней частью, вооруженной крючочками, паразит закрепляется и стенке кишечника. Ядро находится в задней части клетки, длина которой может достигать 16 мм. Длина самых мелких видов не превышает

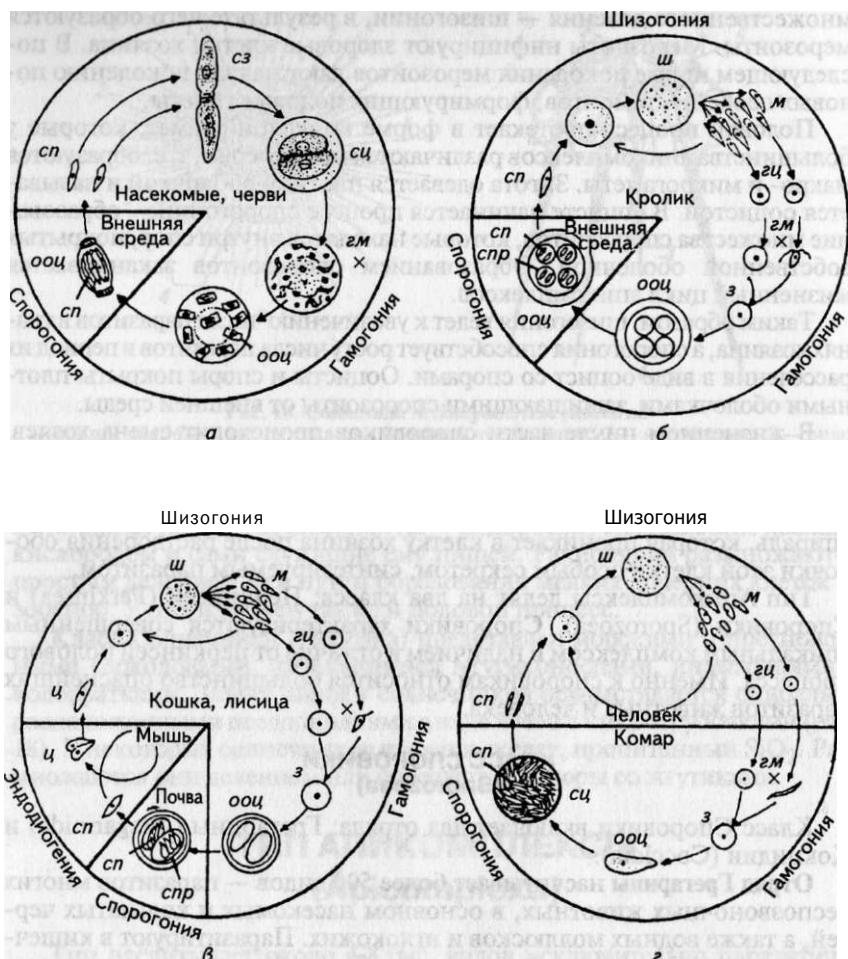


Рис. 19. Схемы жизненных циклов споровиков:
 a — грегарини; b — кокцидии; c — токсоплазмы; d — малярийного плазмодия; gm — гаметы; zg — гаметоциты; z — зигота; m — мерозоиты; oo — ооциста; cs — сизигий; sp — спорозоит; sp — спора; cp — спороциста; w — шизонт; u — цистозоит

15 мкм. Большинство грекарин размножаются половым путем, и лишь у немногих представителей происходит смена полового и бесполого поколений. Так, перед размножением грекарины, паразитирующие в кишечнике жука-чернотелки (рис. 19, 20), соединяются попарно в цепочку (сизигий), округляются и покрываются общей плотной оболочкой, образуя цисту. При этом слияния грекарин внутри цисты не происходит. Ядро каждой особи многократно делится. В каждой особи си-

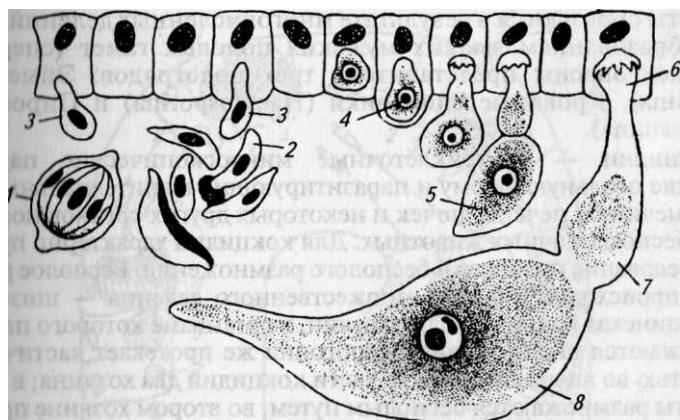


Рис. 20. Схема развития грегарини:
 1 — пора со спорозоитами; 2 — спорозоиты, вышедшие из споры; 3 — спорозоит, внедряющийся в эпителиальную клетку; 4, 5 — развитие спорозоита в грегарине; 6 — эпителий; 7 — протомерит; 8 — дейтомерит

ни ии образуется множество гамет. Микрогаметы имеют жгутики. Клетки, образовавшиеся в разных особях одного сизигия, попарно конкурируют. В результате копуляции гамет образуется зигота, которая окружает гаметы плотной оболочкой и инцистируется, превращаясь в ооцисту. Таким образом в одной цисте заключено несколько ооцист. Цисты, из которых находится множество ооцист, выводятся из тела хозяина по внешнюю среду. Это может произойти и после смерти самого хозяина. Дальнейшее развитие паразита происходит уже во внешней среде и под присутствием кислорода. Внутри каждой ооцисты протекает процесс деления, в результате которого образуется восемь спорозоитов. Эти спорозоиты становятся инвазионными, т. е. в каждой цисте под общей оболочкой находится множество инвазионных ооцист с восемью спорозоитами каждая. Ооцисты заглатываются новым хозяином с загрязненной пищей. В кишечнике хозяина спорозоиты покидают ооцисту, вне которой они в клетки кишечника, развиваются в них, а затем разрывают эти клетки и вырастают во взрослого паразита.

Таким образом, особенность рассмотренного жизненного цикла грегарини заключается в том, что внутри хозяина происходит только половое размножение, в во внешней среде — спорогония с формированием спорозоитов. Хозяин заражается, потребляя пищу, загрязненную цистами. В кишечнике нового хозяина оболочка цисты разрушается, из нее выделяются оболочки ооцист и из них выходят спорозоиты.

Отряд Кокцидии (Coccidia). В отличие от грегарин кокцидиообразные в основном внутриклеточные паразиты. У большинства происходит чередование полового и бесполого размножения. Макрофагамета (женская половая гамета, или яйцо) образуется непосредственно в процессе роста гамонта (гаметоцита) без деления. Микро-

рогаметы образуются в результате многочисленных делений гамонта с образованием мелких мужских половых гамет (спермии). Наиболее опасны представители трех подотрядов: Эймериевые (*Eimeriina*), Кровяные споровики (*Haemosporina*) и Пироплазмы (*Piroplasmida*).

Кокцидии — внутриклеточные микроскопические паразиты, имеющие овальную форму и паразитирующие в эпителиальных клетках кишечника, печени, почек и некоторых других органов позвоночных и беспозвоночных животных. Для кокцидий характерно правильное чередование полового и бесполого размножения. Бесполое размножение происходит в форме множественного деления — шизогонии. У большинства кокцидий один хозяин, в организме которого паразиты размножаются шизогонией. Спорогония же протекает частично или полностью во внешней среде. У части кокцидий два хозяина; в первом паразиты размножаются бесполым путем, во втором хозяине происходит половой процесс и спорогония.

Многие виды паразитических кокцидий из рода *Eimeria* приносят значительный ущерб животноводству. Паразитируя только у позвоночных животных, кокцидии чаще всего поражают кроликов, птиц разных видов, рогатый скот. При этом наиболее подвержен заболеванию молодняк животных. Кокцидии паразитируют в клетках кишечника и вызывают кровавый понос, приводящий к гибели большей части молодняка. Болезнь называется кокцидиозом.

Особый ущерб наносит *Eimeria magna*, поражающая кроликов, в организме которых ооцисты паразита попадают с загрязненными кормом и водой (рис. 21). В кишечнике кролика из ооцист выходят спорозоиты, которые внедряются в клетки стенок кишечника. В этих клетках кокцидии растут и размножаются бесполым путем посредством множественного деления — шизогонии. Дочерние особи носят название мерозоитов. Мерозоиты из пораженных клеток выходят в просвет кишечника, внедряются в здоровые клетки, растут и снова размножаются шизогонией. У *E. magna* развивается пять поколений мерозоитов. Последняя, пятая, генерация мерозоитов в клетках кишечника преобразуется в гамонты. Макрограмбы не делятся и дают начало макрограммам (яйцам), микрограммам путем деления образуют множество подвижных микрограмм с двумя жгутиками (спермии). Микрограммы подвижны и выходят в просвет кишечника. Одна из микрограмм проникает в макрограмму и копулирует с ней. После копуляции гамет образуется зигота. Зигота одевается прочной оболочкой и превращается в ооцисту, которая выводится из кишечника кролика с испражнениями наружу.

Во внешней среде в присутствии кислорода в ооцисте проходит процесс спорогонии: сначала образуется четыре споробласта, которые покрываются собственными оболочками и превращаются в споры. В каждой споре формируется по два спорозоита. По окончании спорогонии споры становятся инвазионными. В каждой инвазионной ооцисте содержится по восемь спорозоитов. Если такая ооциста попадает в ки-

V

Рис. 21. Жизненный цикл кокцидий рода *Eimeria*:

I — мерное поколение шизогонии; **II** — второе поколение шизогонии; **III** — третье поколение шизогонии; **IV** — гамогония; **V** — спорогония; **1** — спорозоиты; **2** — одноядерный макрограмет; **3** — многоядерный шизоит; **4** — образование мерозоитов; **5** — мерозоиты; **f** — расщепление макрограмета; **7** — развитие микрограмета; **8** — ооциста; **9, 10** — образование (НОМ) пластов (видно остаточное тело); **II** — образование спор; **12** — зрелая ооциста с четырьмя спорами, в каждой из которых имеется по два спорозоита

III⁴ и I к кролику, то спорозоиты выходят из споробластов и ооцисты, **Мячики** **новый** цикл развития. Весь цикл развития этого паразита завершается за 7—8 сут, и, если не происходит повторного заражения, то организм **кролика** освобождается от кокцидий. Именно поэтому так важно принимать меры по исключению возможности повторной инвазии.

Для другого вида эймерии (*E. stiedae*) характерен такой же цикл размножения в организме кролика, но этот вид поражает клетки эпителия протоков печени, где протекает несколько циклов бесполого размножения шизогонией. Часто кролики одновременно поражаются двумя формами кокцидиоза — кишечным и печеночным, что увеличивает тяжесть заболевания животных. Каждому сельскохозяйственному животному свой вид кокцидий. Кокцидиозом болеет и человек. Не ме-

Рис. 22. Токсоплазма:
а — токсоплазмы, возникшие в результате продольного деления; б — циклы развития токсоплазм и разные способы заражения ими хозяев; 1 — кошка-хозяин, в которой проходит шизогония и стадии полового цикла; 2, 3, 4 — стадии развития ооцист, в каждой из которых в конечном счете развивается по две споры с четырьмя спорозоитами внутри; 5, 6 — мыши-хозяева, в которых протекает дополнительное бесполое размножение; 7 — внутриутробное заражение мышей

нее опасна кокцидия *E. tenella*, вызывающая опасное заболевание цыплят, приводящее к массовой гибели птицы. Болеют кокцидиозом телята (*E. ztimi*, *E. smithi*), карловые рыбы (*E. carpelli*).

Примерно такой же, как у кокцидий, жизненный цикл характерен для токсоплазмы (*Toxoplasma gondii*), вызывающей опасное заболевание у человека — токсоплазмоз. Однако жизненный цикл токсоплазмы усложнен сменой хозяев и появлением некоторых особенностей в размножении (рис. 22). Основным хозяином этого паразита являются кошки и другие виды кошачьих, в кишечнике которых токсоплазмы растут и размножаются сначала путем шизогонии, а потом и половым путем с образованием ооцист.

Ооцисты с испражнениями животного попадают во внешнюю среду, где в ооцистах происходит процесс спорогонии: в каждой образуются две споры с четырьмя спорозоитами. Инвазионные ооцисты со спорозоитами могут быть заглоchenы с загрязненными пищей и водой промежуточными хозяевами (грызуны, птицы). В кишечнике промежуточного хозяина из ооцист и спор выходят спорозоиты; они внедряются в ткани и проникают в кровяное русло. Паразиты могут оседать в любых тканях, в том числе и в мышцах, где они размножаются бесполым путем. Это особая форма деления, когда две дочерние особи образуются внутри материнского организма. Так возникают скопления паразитов, окруженные собственной общей оболочкой. Эти скопления токсоплазм в промежуточном хозяине носят названия цист.

При поедании зараженной цистами мыши в кишечнике кошки токсоплазмы выходят из цист, внедряются в эпителиальные клетки кишечника и начинают новый жизненный цикл. При общении с больной гоксоплазмозом кошкой человек может также стать промежуточным хозяином. Особенно велика вероятность заражения у детей, играющих и песочницах, где испражняются больные токсоплазмозом кошки. У человека токсоплазмоз протекает в легкой и тяжелой формах. В последнем случае возможен летальный исход.

Таким образом, особенностью токсоплазм является то, что источником инвазии служат не только ооцисты, но и ткани зараженного промежуточного хозяина, содержащие цисты токсоплазм. У млекопитающих токсоплазмы могут передаваться развивающемуся плоду через плаценту; это так называемый врожденный токсоплазмоз. В данном случае плод обычно погибает.

Подотряд Кровяные споровики (*Haemosporina*) представлен большой группой широко распространенных внутриклеточных паразитов крови, часть жизненного цикла которых протекает в эритроцитах млекопитающих и птиц. В отличие от кокцидий у кровяных споровиков спорогония никогда не протекает во внешней среде, а происходит в организме кровососущих насекомых, чаще комаров, которые одновременно являются и переносчиками паразитов. К этим паразитам относится возбудитель малярии, являющийся бичом населения многих тропических и субтропических стран.

В человеке паразитируют четыре вида плазмодия (род *Plasmodium*). Жизненный цикл малярийного плазмодия (*Plasmodium vivax*) типичен для остальных видов. Человек заражается при укусе комаром рода *Anopheles*, который в кровь человека вместе со слюной вносит спорозоитов малярийного плазмодия (рис. 23). С током крови спорозоиты достигают печени, где внедряются в паренхимные клетки печени, превращаются в шизонты и дают первое поколение путем шизогонии. Вышедшие из разрушенных клеток печени мерозоиты проникают в кровь и внедряются в эритроциты, где снова делятся шизогонией. Вышедшие из разрушенных эритроцитов мерозоиты (образуется 10—20 мерозоитов из одного шизонта) снова внедряются в здоровые эритроциты.

Продолжительность одного этапа шизогонии специфична для каждого вида плазмодия. У *PL malariae* промежутки между двумя последовательными бесполыми размножениями составляют 72 ч, поэтому заболевание получило название 4-дневной лихорадки. У наиболее широко распространенного вида *PI. vivax* этот промежуток равен 48 ч; это 3-дневная лихорадка. У *PI. falciparum* срок между двумя размножениями шизогонией составляет тоже около 48 ч, но промежутки между двумя приступами лихорадки сокращаются до 24 ч из-за периода высокой температуры тела у больного человека (тропическая лихорадка). Еще один вид плазмодия встречается лишь в тропической Африке — *PL ovale*.

Выход мерозоитов из эритроцитов сопровождается приступами лихорадки с повышением температуры, так как вместе с мерозоитами из

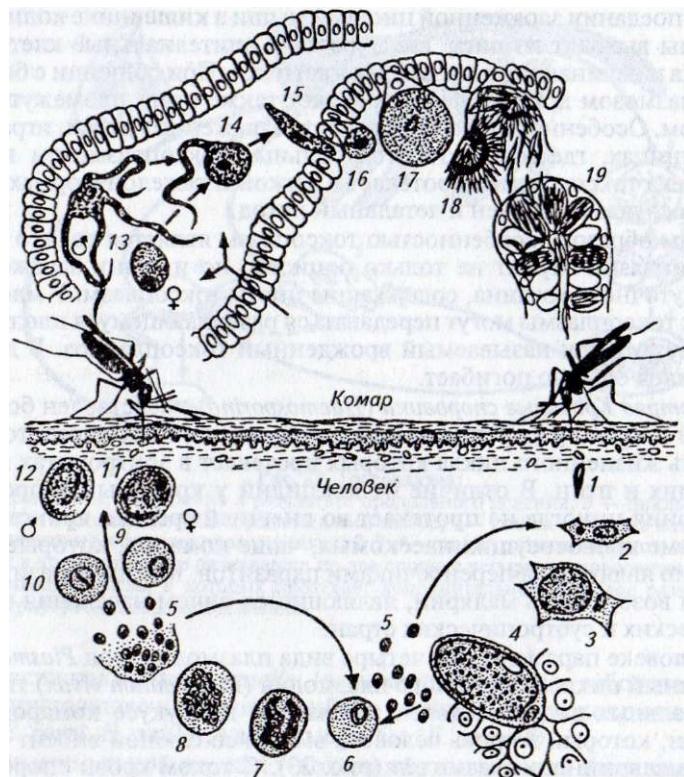


Рис. 23. Жизненный цикл малярийного плазмодия рода *Plasmodium*:
 1 — спорозоит; 2,3 — рост шизонта (агамонта); 4 — шизогония в клетках печени; 5 — мерозоиты; 6,7,8 — шизогония в эритроцитах; 9—12 — образование гамонтов (микро- и макрограмонтов); 13 — образование макрограмм и микрограмм; 14 — копуляция гамет; 15 — зигота (оокинета); 16—18 — спорогония и образование спороцисты со спорозоитами; 19 — накопление спорозоитов в слюнных железах комара

разрушенных эритроцитов в кровь поступают продукты обмена паразитов, вызывающие интоксикацию человека.

После нескольких циклов бесполого размножения шизогонией начинается подготовка к половому процессу. Внедрившиеся в эритроциты мерозоиты дают начало гамонтам, а не шизонтам, как при шизогонии. При этом образуются две группы гамонтов: макрограмм и микрограмм. Дальнейшего развития гамонтов в крови человека не происходит: человек становится носителем малярийного паразита.

У комара, напившегося крови больного малярией человека, в кишечнике из макрограмма формируется женская макрограмма, а из микрограмма образуется четыре—восемь мелких мужских микрограмм. После копуляции макро- и микрограмм образуется подвижная зи-

I o гп — оокинета. Оокинета внедряется в стенку кишечника и на ее внешней стороне в полости тела комара оокинета преобразуется в ооцисты. В ооцисте происходит процесс спорогонии с образованием множества (до 500 особей). Стенки ооцисты разрушаются и спорозоиты выходят из ооцисты и попадают в полость тела комара, откуда они проникают в слюнные железы и в ротные железы. При укусе человека комар со слюной вводит в кровь человека хоботок спорозоиты. Таким образом, в жизненном цикле малярийного плазмодия отсутствуют стадии развития, протекающие во внешней среде. Помимо этого на всем протяжении своего развития паразит не имеет стадий, когда он одевается защитными оболочками, что отличает его от кокцидий.

Малярия широко распространена на планете. У человека, больного малярией, наблюдается малокровие (анемия), интоксикация всего организма; болезнь сопровождается приступами лихорадки. От кровяных паразитов в мире погибло больше людей, чем во всех войнах. Переносит малярию комары рода *Anopheles*, в основном *A. maculipennis*. В Европе существует шесть видов-двойников малярийного комара, которых ранее объединяли в один вид.

Среди кровяных паразитов, вызывающих тяжелые заболевания многих диких млекопитающих и сельскохозяйственных животных, следует отметить пироплазмы (подотряд Пироплазмы — *Piroplasmina*). У них микроскопические паразиты эритроцитов. В эритроцитах пироплазмы размножаются бесполым путем с образованием двух клеток. При этом паразиты не разрушают гемоглобин, который у больных животных появляется в моче.

Жизнь этих паразитов протекает в двух хозяевах: млекопитающих и насекомых (паразитических) клещах. Нападая на больных пироплазмозом животных, клещи вместе с кровью получают и клетки паразита, которые размножаются в теле клещей бесполым путем и проникают в их слюнные железы. При укусе и кровососании клещи передают здоровому млекопитающему пироплазмы. Это заболевание широко распространено во многих странах. Пироплазмы из пораженной самки клеша могут проникать в ее яйца и затем в личинки, а те в свою очередь переносить паразитов млекопитающим.

ТИП МИКСОСПОРИДИИ (*Mixotrichia*)

Известно более 870 видов миксоспоридий, в основном паразитов рыб и малоштамковых червей. В конце своего жизненного цикла микроспоридии образуют споры. Но в отличие от споровиков споры микроспоридий являются многоклеточными образованиями с полярными капсулами, в каждой из которых находится спиральная полярная нить. Кроме того, жизненный цикл миксоспоридий включает развитие паразита от одноядерной фазы к многоядерной. Многоядерная фаза завершается формированием множества многоклеточных спор с двуядерным амебным зародышем. Взрослым паразитам свойствен ядерный дуализм.

Миксоспоридий делят на два класса: класс Собственно миксоспоридии (*Myxosporea*) и класс Актиноспоридии (*Actinosporea*). Актиноспоридии паразитируют в малошетинковых червях и характеризуются некоторыми особенностями строения спор.

Миксоспоридии паразитируют преимущественно в коже рыб (тканевые паразиты); в результате у рыб образуются желваки-опухоли. В этих опухолях находятся взрослые многоядерные плазмодии миксоспоридий. Размеры паразитов колеблются от микрометров до двух сантиметров.

Ядра плазмодиев делятся на вегетативные и генеративные (ядерный дуализм). Вегетативные ядра регулируют обменные процессы в организме паразита, генеративные ядра участвуют в образовании спор, формирование которых происходит внутри плазмодия. Вокруг каждого генеративного ядра образуется обособленная генеративная клетка. В этой клетке ядро неоднократно делится и образуется панспоробласть. Внутри панспоробласта формируются две споры.

Споры миксоспоридий разнообразны по форме, но всегда являются многоклеточными образованиями с полярными капсулами. Так, спора миксоспоридия *Myxobolus* (возбудителя шишечной болезни у рыбы-усача) формируется из шести клеток, что подтверждается наличием шести ядер (рис. 24). Две клетки образуют две створки споры, две другие — капсулы с полярной нитью и две оставшиеся — амебоидный зародыш.

Из больной рыбы споры попадают в воду, где заглатываются другой рыбой. В кишечнике рыбы полярные капсулы выбрасываются и нити вонзаются в стенку кишки. Створки споры раскрываются и амебоидный двуядерный зародыш проникает через эпителий кишки в кровь. По кровяному руслу зародыши попадают под кожу или в другие ткани и органы рыбы. Сначала два ядра в зародыше сливаются и зародыш ста-

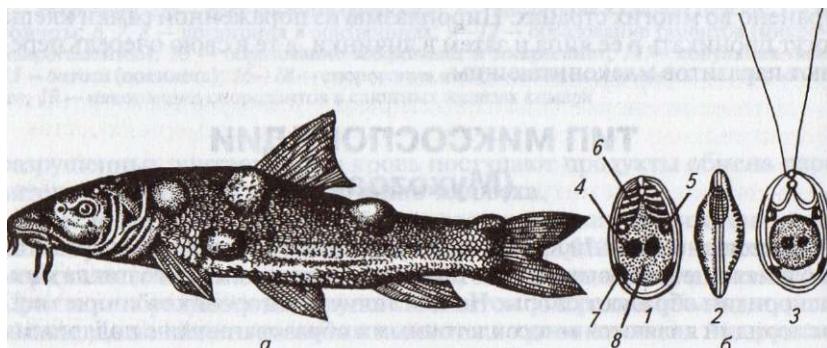
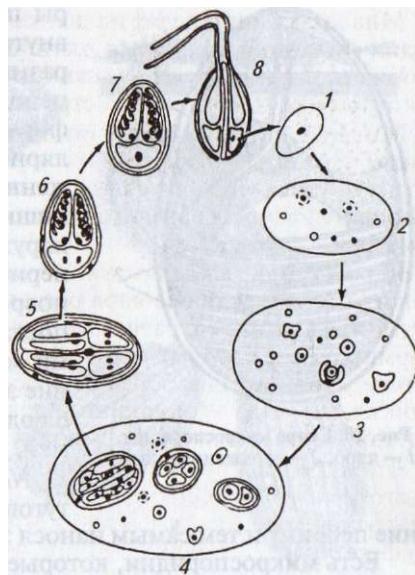


Рис. 24. Слизистые споровики:
а — рыба, пораженная споровиками; б — споры; 1 — спора с неразряженной капсулой;
2 — вид споры сбоку; 3 — спора с разряженной капсулой; 4 — ядро амебоидного зародыша;
5 — ядра клеток — образовательниц полярных капсул; 6 — стрекательная нить;
7 — ядра створок споры; 8 — амебоидный зародыш

I. < с. Лишенный цикл миксоспоридиев
Мухогод:

1 — зародыш с диплоидным ядром; 2 — образование многоядерного мицелия с вегетативными и генеративными ядрами; 3 — формирование спор; 4 — формирование поры с двуядерным амебоидным питомцем; 5 — изгнание спор; 6 — сформированная спора; 7 — образование диплоидного зародыша; 8 — выстrelивание из культических нитей из споры при прорастании.



пищци и я диплоидным. Затем ядрышко и инвагинация и зародыш превращается в многоядерный плазмодий с ядерным дуализмом. После деления в плазмодии образуются нормы (рис. 25).

Миксоспоридии вызывают миеионую гибель многих рыб, при уплотненных поселениях и прудовых хозяйствах. Этот ущерб наносит форели миксоспоридия *Myxosoma cerebralis*, циркляция которой скелет рыбы. У заболевших мальков искривляется позвоночник и нарушается координация движений.

ТИП МИКРОСПОРИДИИ (Microspore)

Существует около 800 видов микроспоридий, которые являются внутренними паразитами насекомых и других беспозвоночных животных; очень немногие виды паразитируют у позвоночных животных.

Микроспоридии из самых мелких простейших организмов, размеры их составляют 4-6 мкм.

Жизненный цикл микроспоридий также заканчивается образованием спор, но эти споры имеют иное строение, чем у споровиков и мицелий. У микроспоридий спора является одноклеточным оболоченным яйцом, в котором имеется одно-два ядра и одна ввернутая полярная нить (рис. 26). У микроспоридий отсутствует половой процесс. Они размножаются бесполым путем, образуя цепочки клеток внутри паразитной клетки хозяина.

Микроспоридия *Nosema apis* наносит серьезный ущерб пчеловодческим хозяйствам. Пчелы заражаются, заглатывая с кормом споры плазмодии. В кишечнике пчелы споры разбухают и выстреливают свои покрытые нити, которые вонзаются в стенку кишечника насекомого. Из спор

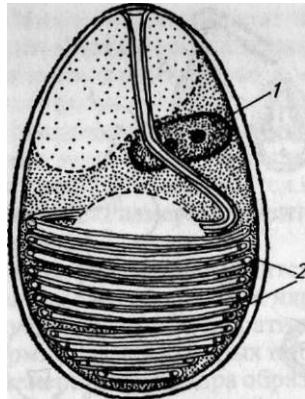


Рис. 26. Спора микроспоридии:
1 — ядро; 2 — заряженная нить

ры по каналу нити зародыш проникает внутрь кишечной клетки. В ней паразит размножается бесполым путем, образуя цепочки клеток. Из этих клеток затем формируются одноклеточные споры с полярной нитью. После разрушения пораженной клетки споры попадают в просвет кишки и с каловыми массами выводятся наружу. Больные нозематозом пчелы и период зимовки пачкают соты калом, одновременно загрязняя их спорами. При поедании пчелами загрязненного спорами меда происходит повторное заражение этих насекомых. При нозематозе наблюдают большой отход пчел и гибель семей.

Nosema bombycis патогенна для гусениц листового шелкопряда, вызывая заболевание пебрину и тем самым нанося значительный ущерб шелководству

Есть микроспоридии, которые, являясь паразитами вредных насекомых, используются в биологической борьбе с вредителями.

ТИП ИНФУЗОРИИ, ИЛИ РЕСНИЧНЫЕ (*Ciliophora*)

Инфузории отличаются наиболее сложной организацией среди одноклеточных животных. Их тело покрыто пелликулой, которая позволяет им иметь относительно постоянную форму. Под пелликулой расположена эктоплазма, в которой находятся многие органеллы, в том числе базальные тельца ресничек (рис. 27); сократительные волоконца — мионемы; защитные органеллы — трихоцисты. При раздражении инфузории выбрасывают из трихоцист множество упругих нитей, которые поражают врага, парализуя его.

Известно более 7,5 тыс. видов инфузорий, населяющих моря и пресные водоемы и входящих в состав планктона, бентоса и детрита; некоторые виды обитают в почве. Среди инфузорий много хищных и паразитических форм, в кишечнике жвачных животных обитают симбиотические формы. Органеллами движения инфузорий служат многочисленные реснички. При этом подавляющее большинство этих животных обладает ресничками в течение всей жизни. Реснички инфузорий по своему строению сходны со жгутиками. Ресничный аппарат весьма разнообразен. Особенно сложный ресничный аппарат расположен около рта.

Вторым отличительным признаком представителей этого типа является присутствие в их теле двух ядер (ядерный дуализм): крупного вегетативного ядра (макронуклеуса) и значительно более мелкого генеративного ядра (микронуклеуса).

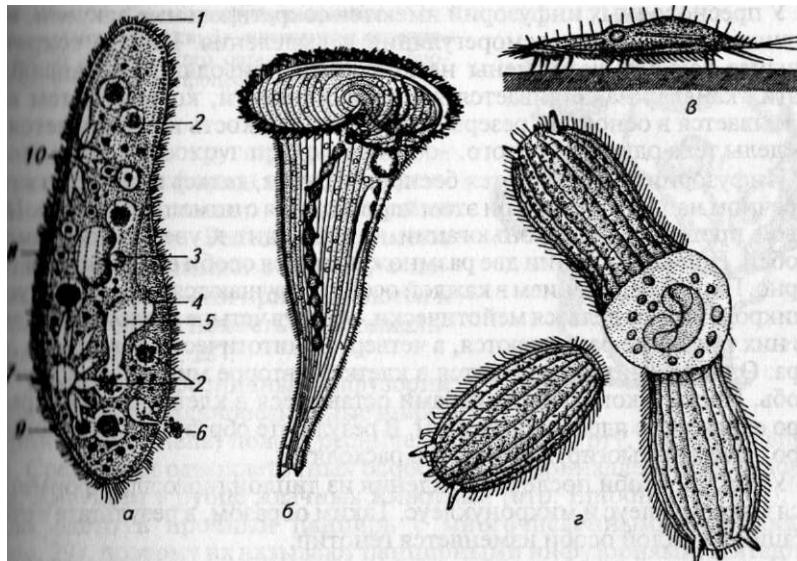


Рис. 27. Инфузории:
а — усач-лька; **б** — трубач; **в** — брюхоресничная инфузория, ползающая с помощью утолщенных щетинок; **г** — хищные инфузории колепсы, напавшие на другую инфузорию;
1 — нгредний конец; **2** — пищеварительные вакуоли; **3** — микронуклеус; **4** — реснички;
5 — кусочная глотка; **6** — удаленная через порошицу непереваренная пища; **7** — сократимый вакуоль; **8** — макронуклеус; **9** — трихоцисты; **10** — приводящие каналы сократимой вакуоли.

Многие инфузории обладают сложной системой пищеварения. Расположенный в углублении тела (перистоме) рот, или цитостом, окружён длинными ресничками, с помощью которых пищевые частицы и погружаются в него. Часто рот ведет в глотку, погруженную в эндоплазму. В эндоплазме пищевые частицы окружены пузырьками, содержащими пищеварительные ферменты, в результате образуются пищеварительные вакуоли. В пищеварительных вакуолях создается кислая среда. На последующих этапах переваривания пищи среда становится нейтральной, что аналогично процессам пищеварения у высших животных. Непереваренные остатки пищи выбрасываются из тела инфузории в определенном месте через порошицу. Есть виды хищных инфузорий, вооруженных ротовым хоботком, с помощью которого они проникают покровы своей жертвы.

Большинство инфузорий питаются бактериями, немногие поедают одноклеточные водоросли, и среди них встречаются даже монофиты. Хищные инфузории порой охотятся на жертв, которые по размерам больше хищниц; это, например, инфузория-туфелька. Жертву они поражают хоботком и высасывают ее содержимое. Свободноживущие инфузории являются важным звеном в пищевых цепях экосистем.

У пресноводных инфузорий имеются сократительные вакуоли, выполняющие функции осморегуляции и выделения. Иногда сократительные вакуоли усложнены несколькими приводящими каналами. В этих каналах накапливается избыток жидкости, которая затем выбрасывается в основной резервуар; из него жидкость выталкивается за пределы тела одноклеточного.

Инфузории размножаются бесполым путем, делясь надвое, но в перечном направлении. При этом ядро делится с помощью митоза. Половой процесс в виде конъюгации не приводит к увеличению числа особей. При конъюгации две размножающиеся особи соединяются попарно. Перед соединением в каждой особи разрушаются макронуклеусы, а микронуклеусы делятся мейотически, образуя четыре гаплоидных ядра. Из них три также разрушаются, а четвертое митотически делится на два ядра. Одно из этих ядер остается в клетке, а второе мигрирует в другую особь. После такого обмена ядрами оставшееся в клетке стационарное ядро сливаются с ядром-мигрантом. В результате образуется диплоидное ядро. Затем конъюгирующие особи расходятся.

У каждой особи после расхождения из диплоидного ядра формируются макронуклеус и микронуклеус. Таким образом, в результате конъюгации в каждой особи изменяется генотип.

При классификации инфузорий в качестве диагностических признаков используют особенности строения ротового аппарата или структуру ресничного аппарата. Последний подход преобладает. Инфузорий делят на два класса: класс Ресничные инфузории (*Ciliata*) и класс Сосущие инфузории (*Suctoria*).

Класс *Ресничные инфузории* наиболее многочисленный. Представители этого класса покрыты ресничками на протяжении всех стадий жизненного цикла. Среди представителей подкласса Равноресничные инфузории (*Holotricha*), характеризующихся равномерным расположением на теле ресничек равной длины, много свободноживущих (например, инфузория-туфелька, *Paramecium caudatum*), хищных, питающихся своими собратьями, и паразитических форм. Среди последних следует отметить инфузорию балантидий (*Balantidium coli*), которая встречается в кишечнике свиней и человека. Эта инфузория питается в основном содержимым кишечника, но может разрушать слизистую кишечника, вызывая заболевание — балантидиоз. Заражение происходит при потреблении загрязненного цистами балантидия пищи и воды.

В природных водоемах и в прудовых хозяйствах, занимающих разведением рыб, большой вред наносят паразитические инфузории. Например, равноресничная инфузории *Ichthyophthirius* внедряется в кожу рыб и начинает питаться клетками хозяина. В результате на теле рыбы образуются многочисленные язвочки. Заболевание может привести к гибели рыб, особенно молоди карпа. На жабрах и коже часто паразитируют инфузории из рода *Trichodina*, причиняя молоди рыб существенный вред.

У большинства кругоресничных инфузорий (подкласс Кругоресничные инфузории — *Peritricha*) реснички располагаются левоспир-

Рис. 28. Сидячие инфузории сувойки:
1, J — деление сувойки; 3 — плавающая, отделившаяся от материнского организма клетка-бродильник; 4 — половой процесс

ришно только вокруг предротовой воронки. Многие формы ведут прикрепленный образ жизни. Так, Сувойки (*Vorticella*, отр. Peritrichida) имеют винный сократимый стебелек, с помощью которого они прикрепляются к субстрату. Среди них есть и колониальные формы (рис. 28).

У спиральноресничных инфузорий (подкласс Спиральноресничные инфузории — Spirotricha) полоса ресничек, ведущих ко рту, закручена вправо. Среди этих одноклеточных особое место принадлежит инфузориям, живущим в рубце жвачных животных (отр. Entodiniomorpha). Их тело одето в прочный панцирь с многочисленными отростками (рис. 29), поэтому их называют панцирными инфузориями. Питаются они бактериями рубца и способствуют расщеплению клетчатки коры, взаимодействуя сложным образом с целлюлозорасщепляющими микробами. Эти полезные симбионты не только участвуют в переваривании пищи, но и сами служат источником питания для жвачных «и потных».

Нестейшие из класса Сосущие инфузории (Suctoria) лишены ресничек и в большей части жизненного цикла, и лишь на ранних этапах размножения дочерняя клетка — бродяжка — имеет реснички. У них нет рта и околосротовой воронки. Эти инфузории с шаровидным телом, на котором индивидуально расположены щупальца (рис. 30). Щупальца служат для прикрепления к субстрату и, кроме того, наполняются ловчим аппаратом. Поймав мелких инфузорий с помощью липкого секрета, суктории как бы перекачивают содержимое жертвы в свое тело.

При бесполом размножении от материальной клетки суктории отпочковываются дочерняя особь — бродяжка,

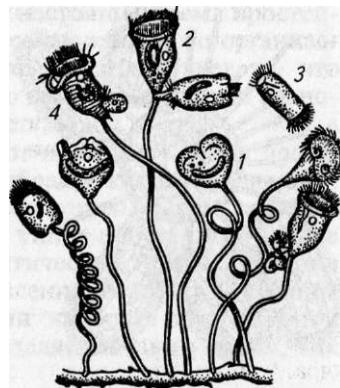


Рис. 24. Инфузория из желудка жвачного млекопитающего:
1 — перепонки, загоняющие пищу в рот; 2 — клеймо плотка; 3 — реснички; 4 — сократительные никуоли



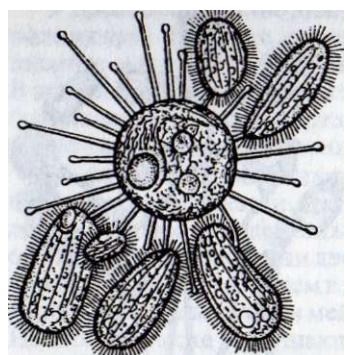


Рис. 30. Сосущая инфузория *Sphaerophrya*, высасывающая щупальцами содержимое нескольких ресничных инфузорий

покрытая ресничками. В последующем она теряет ресничный аппарат и превращается в сукторио с щупальцами. Следует отметить важную роль некоторых мелких видов инфузорий, жгутиконосцев и амеб в жизни почвы. Бактерии, в том числе азотфикссирующие, служат основной пищей для простейших. Однако они не только поедают бактерий, но и способствуют их размножению, выделяя в почву вещества, стимулирующие этот процесс. Простейшие способствуют повышению плодородия почвы, активно участвуют в процессах биологической очистки водоемов.

Имеется опыт искусственного разведения паразитических простейших для борьбы с вредными насекомыми.

ФИЛОГЕНИЯ И ЭКОЛОГИЧЕСКАЯ РАДИАЦИЯ ПРОСТЕЙШИХ

Мир живых существ делят на два надцарства: Безъядерные, или Прокариоты (Prokaryota), и Ядерные, или Эукариоты (Eucaryota).

Клетки прокариот в отличие от клеток эукариот не имеют оформленного ядра. Эукариот обычно делят на три царства: Растения (Vegetabilia, или Plantae), Грибы (Mycetalia, или Fungi) и Животные (Animalia, или Zoa). Большинство растительных организмов — автотрофы, самостоятельно синтезирующие органические вещества в процессе фотосинтеза. Грибы, хотя и относятся к гетеротрофным организмам, но питаются растворенными органическими веществами. Животные являются гетеротрофами, существующими за счет потребления других организмов или их остатков. Но различия по типам питания между этими царствами относительны, так как имеется множество переходных форм, особенно среди низших представителей.

Простейших (Protozoa) относят к примитивным одноклеточным эукариотам. Признано, что эукариоты берут начало от прокариот. Косвенным свидетельством их единства служит сходство процессов синтеза белка в клетке. Безъядерные организмы были одними из первых на нашей планете, часть из них способна существовать даже в бескислородной среде.

Среди эукариотических животных клеточный уровень организации рассматривают как более примитивный. Это позволяет полагать, что простейшие на Земле появились первыми и послужили началом более слож-

УttR форм — многоклеточным животным. В настоящее время простейшие существуют во взаимодействии с более сложными по организацией многоклеточными организмами. Этому способствует то, что Бритайшие в процессе эволюции отлично приспособились к различным условиям жизни на нашей планете,

и Полагают, что эукариоты произошли от прокариот путем постепенного возникновения органелл из мембранных клетки прокариот. В эволюции эукариот, по-видимому, большую роль сыграл симбиоз различных прокариот. Из известных семи типов простейших четыре являются исключительно паразитическими группами, которые значительно позднее, после появления их хозяев — высших проклеточных. Поэтому при выяснении наиболее примитивной группы среди простейших все внимание уделяется таким типам, как Саркомастигофоры и Инфузории.

Инфузории обычно относят к самым высокоорганизованным одноклеточным животным: есть представители с чертами многоклеточности. Таким образом, только саркомастигофоры несут первичные приемки, имеющие сходство с предками всех простейших. Но саркомастигофоры чрезвычайно неоднородны, что подтверждается особенностями (группами) подтипов Саркодовые и Жгутиконосцы.

Кольшинство исследователей придерживаются мнения Пашера (1914) о том, что жгутиковых следует относить к более древней группе. У них много общего с одноклеточными растениями, разнообразнее типов питания, а их органеллы движения имеются даже у части прокариот. Можно предположить, что разнообразные способы питания у жгутиконосцев могли стать основой для последующего их разделения на автотрофов и гетеротрофов. Следует отметить, что жгутики есть у гамет иллюзии и Metazoa. Упрощенность (отсутствие жгутиков и пелликулы) саркомастигофоры можно отнести к вторичным явлениям в связи с переходом к особому типу питания — фагоцитозу. Наличие же жгутиконосцев у гамет части саркомастигофоры может лишь свидетельствовать об их происхождении от жгутиконосцев. Кроме того, в последнее время было показано, что у некоторых амеб с ростом происходит редукция жгутиков у взрослых форм. Описано немало форм саркомастигофор, обладающих новидоподиями и жгутиками одновременно.

Таким образом, можно предположить, что предками современных инфузорий были древние Саркомастигофоры с разными способами питания и имеющие примитивные жгутики.

Бесспорно, что Apicomplexa и Ciliophora имеют родство со жгутиконосцами. Споровики могли упроститься в связи с переходом к паразитическому образу жизни, но одновременно с этим у них усложнился жизненный цикл, который включает стадию гамет, снабженных жгутиками. Мухозоа и Microspora могли произойти от древних саркомастигофор, поскольку их развитие начинается с амебоидного иродыша, а гаметы со жгутиками отсутствуют. Возможны варианты пинтономности их эволюционного развития. На основе изложенного выше материала представляется возможной филогенетическая



Рис. 31. Филогения Protozoa

схема Protozoa, представленная на рис. 31.

На базе современных представлений можно обозначить основные пути экологической радиации одноклеточных животных (рис. 32). Центральной группой могли быть многочисленные и разнообразные представители Саркомастигофор, но с преобладанием жгутиковых форм и господством водных форм. В последующем саркодовые утратили жгутики и осуществили переход к ползающему образу жизни и питанию путем фагоцитоза и пиноцитоза. Бентосный образ жизни способствовал образованию защитных раковин разнообразной конструкции, таких как раковины корненожек, фораминифер. У части саркодовых

наружный скелет усложнился и стал более легким, в результате чего они смогли перейти к планктонному образу жизни (радиолярии, солнечники).

В связи с прогрессом клеточного строения возникают крупные паразитарные формы — инфузории. Они активно передвигаются, ведут разнообразный образ жизни. Среди них плавающие, ползающие, сидячие, скважники. Позже возникают симбиотические и затем паразитические формы инфузорий. Последние и поныне продолжают свое развитие, усложняя и совершенствуя отдельные этапы своего жизненного цикла.

Многим простейшим, в основном обитающим в пресных водоемах и паразитирующими в других организмах, свойственно образование цист и спор при наступлении неблагоприятных условий (высыхание и вымерзание водоемов). У морских представителей инцистирование является лишь исключением. В виде цист и спор простейшие могут переноситься ветром, птицами и другими животными на большие расстояния, благодаря чему пресноводные простейшие встречаются на земной поверхности повсюду, где имеются условия для их существования.

Пресноводные простейшие заселяют все виды водоемов, даже самые мелкие лужи или скопления воды, образующиеся в пазухах листьев растений. Множество их и в болотах, где в основном встречаются раковинные амебы. По составу фауны простейших относительно точно можно определить степень загрязнения водоема, так как в зависимости от этой степени водоемы заселяют разные представители простейших; их состав меняется по мере возрастания или снижения степени загрязнения воды. Простейшие живут даже в горячих источниках с темпера-

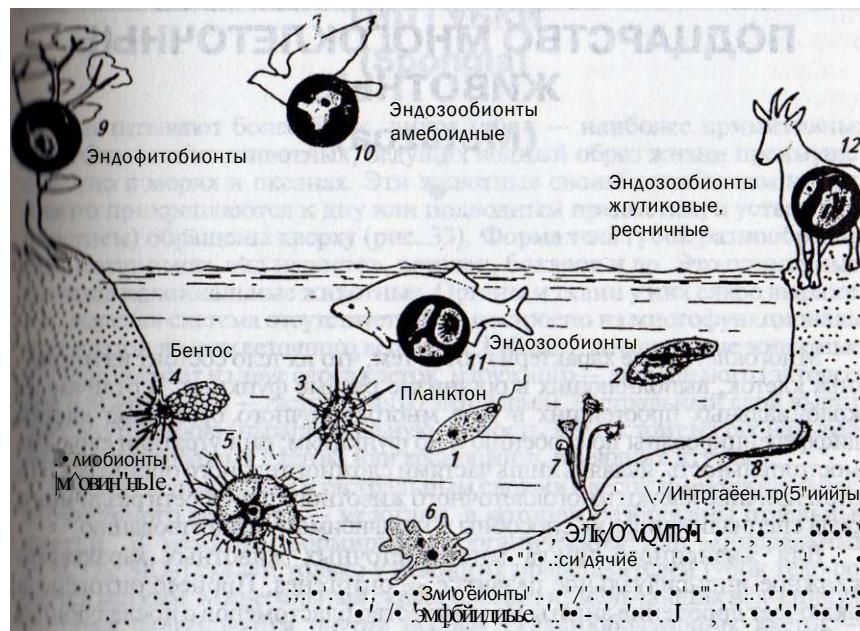


Рис. 32. Экологическая радиация простейших:

1—ушконосец; 2—инфузория; 3—радиолярия; 4—раковинная амеба; 5—фораминыф; 6—амеба; 7—суворка; 8—инфузория; 9—лептомонас; 10—кишечная амеба; II, I/—различные паразитические простейшие

И в приподнятых поди до +50 °C, находят их и в водоемах с высоким уровнем застенности (до 25 %).

Наиболее разнообразен состав простейших в морях и океанах, где они **широко распространены** в огромных количествах всю толщу воды. Вместе с одноклеточными водорослями простейшие служат кормом для других животных, в **числе которых** для рыб и китообразных. Раковинные саркодовые, отмирая, опускаются на дно, образуя мощнейшие донные отложения известняков.

ПОДЦАРСТВО МНОГОКЛЕТОЧНЫЕ ЖИВОТНЫЕ (Metazoa)

Многоклеточные характеризуются тем, что их тело состоит из множества клеток, выполняющих в организме разные функции. В отличие от колониальных простейших в теле многоклеточного организма клетки дифференцированы по строению и по функциям; они утратили свою самостоятельность, являясь лишь частями сложного единого организма. По этой причине клетки многоклеточного животного, приобретя различную роду специализацию, не способны к независимому существованию.

Для жизненного цикла многоклеточных животных характерно сложное индивидуальное развитие — онтогенез. Процесс онтогенеза включает дробление зиготы на множество бластомеров с последующей дифференцировкой их на зародышевые листки и зародышевые листки и зачатки органов, развитие, рост и образование взрослого организма. Увеличение размером тела многоклеточных по отношению к их поверхности способствовало усложнению процессов обмена веществ, что, в свою очередь, обеспечило многоклеточным животным устойчивость жизненных процессов и способствовало продлению их жизни.

Большинство ученых считают, что многоклеточные произошли от Protozoa. В пределах Protozoa прослеживается тенденция перехода к многоклеточности. В отдельных случаях у Protozoa наблюдается даже многоклеточность отдельных фаз развития (Миксоспоридии). Структурные компоненты клеток Protozoa идентичны таковым клеток Metazoa. Важное значение в решении вопроса о происхождении многоклеточных многие исследователи приписывают колониальным простейшим, например *Volvox*, у которого имеются клетки двух типов — соматические и половые.

Еще в 1874 г Э. Геккель утверждал, что предком многоклеточных была шаровидная колония какого-то простейшего, и что в процессе эволюции (филогенеза) за счет втячивания одной половины шара могла возникнуть первичная кишечная полость и первичный рот. Такой уже двухслойный организм плавал с помощью жгутиков, размножался половым путем и впоследствии стал предком многоклеточных.

Существует множество других теорий, но большая часть их сходится в одном: отдаленными предками многоклеточных животных были колониальные простейшие организмы.

ТИП ГУБКИ (*Spongia*)

Нашитмпают более 5 тыс. видов губок — наиболее примитивных Milt мок неточных животных, ведущих водный образ жизни преимущественно и морях и океанах. Эти животные своим основанием неподвижно прикрепляются к дну или подводным предметам, а устьем (отверстием*) И нем) обращены кверху (рис. 33). Форма тела губок разнообразна: они могут иметь вид наростов, веточек, бокалов и др. Это одиночные, Но чище колониальные животные. Органы и ткани у них слабо выражены* • пгтк и межклеточного вещества. Губки двухслойные животные. И*ично состоит из двух слоев клеток: наружного дермального (эктодермы) и внутреннего — гастрального (энтодерма). Гастральный слой выстилает иногреннюю (парагастральную) полость. Он состоит из так называемых ворсинок п шчковых клеток, или хоаноцитов, имеющих жгутики.

Между дермальным и гастральным слоями клеток имеется слой бесструктурного упрочняющего вещества — мезоглея, в котором разбросаны отдельные минеральные частицы. В мезогле формируется органический или минеральный ((SiO_2) скелет; только у немногих представителей губок тело освобождено от скелета. Минеральный скелет состоит из мельчайших игл-спикул, которые формируются внутри особых скелетообразующих клеток — хондроцитов.

Многочисленные, расположенные в мезогле. Роговой (спонгиновый) скелет имеет химическому составу близок к шелку.

Внешняя поверхность тела губок пронизана множеством пор, через которые вода поступает в систему каналов и камер. Движение воды обеспечивается жгутиками хоаноцитов. Из каналов и камер вода попадает в центральную или гастральную полость, откуда выводится наружу через устье.

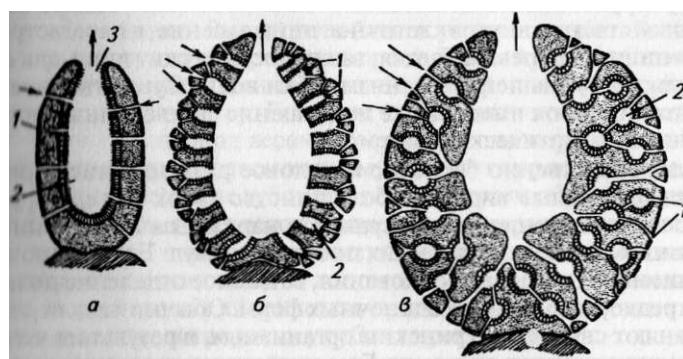


Рис. II. Типы строения губок с различной сложностью системы каналов и расположением жгутиковых камер:

Л — шишкообразная губка; б — сикон; в — лейкон; 1 — поры; 2 — жгутиковые камеры; 3 — устье (стрелки указывают направление тока воды в теле губки)

Специальные клетки — пороциты — способны к сокращению и могут открывать и закрывать поры.

Среди клеток, находящихся в мезоглее, выделяются подвижные клетки — амебоциты. По своим функциям они универсальны: выполняют транспортную функцию, выделяют вещество мезоглея, из них образуются половые клетки и клетки всех других типов, они участвуют в бесполом размножении, дают начало клеткам, образующим скелет жи вотного. Они переносят пищевые частицы от ханоцитов к другим клеткам, удаляют экскреты, а в период размножения переносят спермин в мезоглее к яйцеклеткам. Такие разнообразные функции амебоцитов характеризуют губок как животных, находящихся на эволюционной лестнице ниже прочих многоклеточных.

Наиболее простой тип строения губок называют аскон (рис. 33), но он в основном характерен для одиночных форм и для молодых колониальных особей. Усложнение в период индивидуального развития приводит к возникновению более сложной формы — типу сикон. Дальнейшее усложнение строения тела губок (утолщается мезоглея, образуются карманы и камеры, покрытые слоем воротничковых клеток — ханоцитов) ведет к самому сложному типу — лейкон. Таким образом, у губок типа лейкон и частично сикон парагастральная полость (в отличие от типа аскон) оказывается выстиланной клетками эктодермы. Ими же выстиланы приводящие и отводящие (у лейкон) каналы, которые являются впаяваниями эктодермы. Энтодермой выстиланы лишь жгутиковые камеры, число которых у губок лейконового типа огромно — до нескольких миллионов.

Движение воды по каналам тела губок обеспечивает их организм кислородом и способствует удалению из тела продуктов обмена. С водой в тело губок попадают пищевые частицы (мелкие водные животные и растительные организмы, гниющие остатки), которые захватываются псевдоподиями ханоцитов и перевариваются в их цитоплазме. Часть захваченной пищи передается амебоцитам, и те ее переваривают и транспортируют питательные вещества в мезоглею. Таким образом, губкам свойственно внутриклеточное пищеварение, в парагастральной полости пища не переваривается; эта полость служит лишь для сбора и эвакуации поступившей в организм губки воды. Существенное значение в питании губок имеет также поглощение растворенных органических веществ осмотическим путем.

Губкам свойственно бесполое и половое размножение. Среди них есть раздельнополые виды, но большинство губок — гермафродиты. Бесполое размножение осуществляется наружным почкованием или образованием особых внутренних почек — геммул. Если бесполое размножение носит характер почкования, то полное отделение почки происходит редко, в основном у одиночных форм. Обычно же дочерние особи сохраняют связь с материнским организмом, в результате чего образуются и разрастаются колонии. Границы между отдельными особями могут исчезать, и тогда вся колония сливаются в общую массу (рис. 34).

Геммулы в виде групп клеток, окруженных оболочками и содержащих запас питательных веществ, образуются в мезоглее. Эти образова-

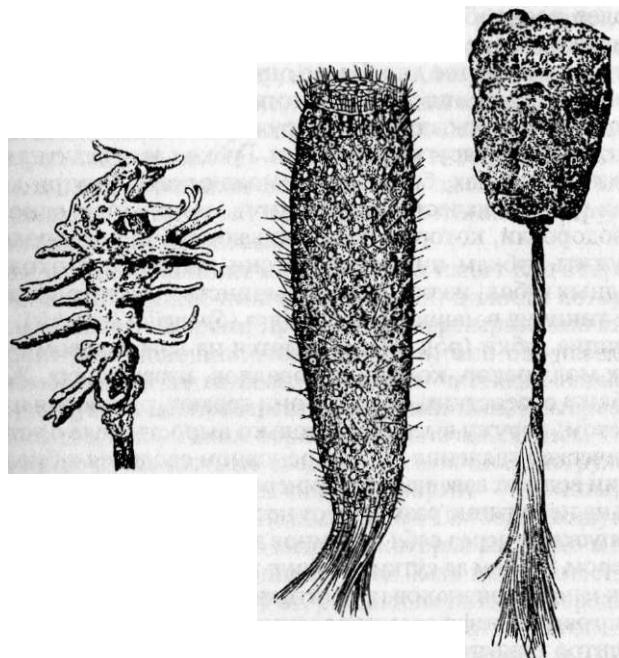


Рис. 34. Различные виды губок:
а — бадяга; б — стеклянные губки

ним представляют собой покоящиеся зимние почки. Так, например, и **р (и овод ная** губка бадяга летом размножается обычным почкованием и **половым** путем. Но к осени в мезоглее бадяги амебоциты образуют **Шаровидные** геммулы. Зимой тело бадяги умирает и распадается. Геммулы **остаются** на дне и перезимовывают. Весной содержащаяся внутри геммул клеточная масса выползает, прикрепляется к субстрату и **ршииняется** в новую губку. Геммулы выполняют также функцию **распрггин**, так как в период весеннего разлива они переносятся течением. **Мри** пересыхании водоемов геммулы могут переноситься ветром.

Ноловое размножение губок происходит путем образования в мезогле амебоцитов яйцеклеток и спермиев. Спермин выносятся в по-
ш п. канальцев и с водой парагастральной полости — во внешнюю ир#лу через устье (оскулюм). С током воды спермин попадают в тело **Уок и, и меющей** зрелые яйцеклетки, проникают в мезоглею и сливаются с ними, т.е. оплодотворение у губок перекрестное.

В материнском организме из зиготы развивается личинка, покрытая ресничками; личинка выходит наружу, активно плавает, перемещаясь течением воды на значительные расстояния, затем опускается на субстрат и прикрепляется к нему и превращается в губку.

Наиболее разнообразны и многочисленны губки тропических и субтропических морей. Встречаются губки на небольших глубинах, предпочитая каменистое дно. Часто они сожительствуют с другими организмами, вступая с ними в симбиотические отношения различного типа. В колониях губок можно обнаружить кольчатых червей, ракообразных, иглокожих и других животных. Губки часто селятся на подвижных животных (крабах, брюхоногих моллюсках). Внутри клеток пресноводных губок в качестве симбионтов часто живут одноклеточные зеленые водоросли, которые обеспечивают губку кислородом и могут также служить губкам пищей. В России встречается около 20 видов пресноводных губок, из которых большинство обитает в озере Байкал. Наиболее типична в наших реках бадяга (*Spongilla lacustris*).

Сверлящие губки (род *Cliona*) селятся на известковом субстрате — раковинах моллюсков, колониях кораллов, известняках. Живут сверлящие губки в отверстиях, которые они делают, растворяя известок особым секретом; наружу выступают только выросты тела с устьями.

Практическое значение губок в основном сводится к биологической фильтрации воды от взвешенных минеральных и органических веществ. Несмотря на небольшие размеры (от нескольких миллиметров до 1,5 м), губки пропускают через себя огромное количество воды: одна губка бадяга размером 5—7 см за сутки профильтровывает около 3 л воды.

У губок много признаков примитивности организации: у них отсутствуют настоящие дифференцированные ткани и органы, для клеточных элементов характерна высокая пластичность и т. п. Губки способны к регенерации: при удалении отдельных участков тела происходит их восстановление. Если измельченную губку просеять через сито, то образовавшаяся масса из отдельных клеток и их групп способна к восстановлению целого организма. Клетки кашицы активно двигаются и собираются вместе, в последующем из этого скопления клеток формируется маленькая губка. Такой процесс формирования организма из скопления клеток называют *соматическим эмбриогенезом*.

Губки — древние организмы. Отделение губок от ствола многоклеточных произошло очень давно. Существует мнение, что губки могли произойти от колониальных воротничковых жгутиконосцев независимо от прочих многоклеточных. Не менее обоснована гипотеза, что многоклеточные произошли общим стволом, от которого одними из первых отделились губки. Вторая гипотеза представляется более обоснованной, ибо личинки губок сходны с личинками планулами кишечнополостных.

ТИП КИШЕЧНОПОЛОСТНЫЕ (*Coelenterata*)

Общая характеристика. Тип объединяет более 10 тыс. видов примитивных многоклеточных животных, ведущих исключительно водный образ жизни и обитающих в основном в морях. Часть из них ведут свободноплавающий образ жизни, другие — сидячий и прикрепленный к дну.

Кишечнополостным свойственна радиальная симметрия, что связано с их образом жизни. У сидячих форм один полюс тела обычно служит для прикрепления к субстрату, на другом имеется рот. Многие органы получают одинаковое развитие, что приводит к радиальной симметрии. Кишечнополостные — двухслойные животные: у них формируются только два зародышевых листка — эктодерма и энтодерма. Между этими листками находится первичная полость тела, заполненная мезоглеей, которая у одних представителей имеет вид пластинки, а у других — это большая масса студенистого вещества.

В простом случае тело кишечнополостных имеет вид открытого на одном конце мешка, в кишечной (гастральной) полости которого, выстланной клетками энтодермы, происходит переваривание пищи. Отверстие служит для кишечнополостных ртом, оно окружено венцом щупалец, помогающих захватывать пищевые частицы. Анальное отверстие отсутствует, а непереваренные остатки пищи выбрасываются через ротовое отверстие. Таким образом, можно заключить, что просто устроенные кишечнополостные сводятся к типичной гаструле. К этой схеме строения наиболее близки сидячие формы — полипы, широко распространенные среди кишечнополостных. Свободноживущие формы имеют уплощенное тело; это медузы, которые активно и пассивно с: течениями передвигаются в водной среде. Тело медуз имеет вид прозрачного студенистого зонтика. Рот, расположенный посередине нижней стороны купола и окруженный предротовыми лопастями, ведет в кишечную полость, от которой отходят радиальные каналы. Океанические медузы достигают двух метров в диаметре.

Деление кишечнополостных на полипы и медузы чисто морфологическое, поскольку иногда один и тот же вид кишечнополостных на разных стадиях жизненного цикла может иметь строение то медузы, то полипа. Медузы — обычно одиночные свободноживущие животные, а полипы в своем большинстве — колониальные формы. Начиная жизнь как одиночный организм, полип путем неполного почкования образует колонии, насчитывающие тысячи особей.

Для кишечнополостных характерно наличие стрекательных клеток, служащих для добывания пищи и защиты.

Размножаются кишечнополостные бесполым (почкованием) и половым путем. У многих форм при этом наблюдается чередование поколений: бесполое поколение полипов сменяется половым поколением медуз.

Строение и жизненные отправления. Покровы кишечнополостных образованы однослойным эпителием эктодермального происхождения. В эпителии располагаются узкоспециализированные клеточные элементы. Это эпителиально-мышечные клетки, содержащие миофибриллы, которые обеспечивают укорочение тела полипа. По всей поверхности тела и особенно густо на щупальцах и вокруг рта разбросаны чувствительные клетки, выполняющие функции рецепторов, мое принимающих сигналы из внешней среды. Характерны в покровах кишечнополостных стрекательные клетки, в основном расположены

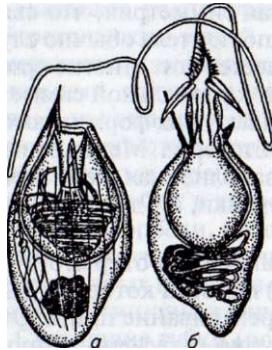


Рис. 35. Стрекательные клетки гидры *Hydra olidactis*:
а — в покоящемся состоянии;
б — с выброшенной нитью

ные на щупальцах (рис. 35). Внутри каждой такой клетки имеется капсула со спирально закрученной полой нитью. Если прикоснуться к чувствительному волоску клетки, стрекательная нить выворачивается и выбрасывается наружу. Вооруженная шипиками нить вонзается в тело жертвы и удерживается в ранке, вводя при этом в нее ядовитый секрет, парализующий мелкую добычу. У крупных животных этот секрет вызывает ожоги. Стрекательные клетки — одноразовое оружие. На месте сработавших клеток образуются новые, так как в покровах кишечнополостных имеются особые клетки, которые могут превращаться в стрекательные, половые, чувствительные и другие.

Нервная система у полипов представлена нервным сплетением диффузного

типа, образованным звездчатыми нервными клетками, соединенными своими отростками. Нервное сплетение лежит под покровным эпителием. У свободноживущих медуз нервная система сложнее: это нервное кольцо, располагающееся по краю купола и скопления нервных клеток вокруг глазков и статоцистов.

Органы чувств примитивны и лучше развиты у медуз (статоцисты и глазки). Чувствительные клетки имеются в покровах тела, особенно на щупальцах и вокруг ротового отверстия.

Мускулатура. У полипов форма тела меняется в результате действия эпителиально-мышечных клеток, имеющих миофибриллы. У медуз движение обеспечивается специальными мышечными волокнами, залагающими в мезоглее по краям купола. У коралловых полипов продольные и поперечные мышечные волокна находятся в перегородках кишечной полости.

Органы пищеварения. У гидр и близких к ним форм ротовое отверстие открывается непосредственно в кишечную (гастральную) полость. У большинства же видов рот ведет в эктодермальную глотку и затем в кишечник. У коралловых полипов для увеличения всасывающей поверхности в кишечную полость вдаются радиально расположенные продольные перегородки. У медуз от кишечной полости внутрь купола отходят радиальные каналы, впадающие в кольцевой канал. Кишечная полость у медуз продолжается и в полости щупалец.

Кишечная полость у кишечнополостных выстилана однослойным энтодермальным эпителием, клетки которого имеют жгутики, служащие для перемещения пищевых частиц. Есть особые железистые клетки. Некоторые клетки эпителия образуют псевдоподии, захватывающие частицы пищи. Одновременно с внутриклеточным пищеварением у кишечнополостных частично происходит и полостное переваривание

в кишечной полости с помощью пищеварительных ферментов, вырабатываемых железистыми клетками кишечного эпителия. У гидроидных полипов существуют две фазы переваривания пищи. Сначала они заглатывают крупный комок пищи или целое животное, которое начинает перевариваться в гастральной полости. Затем мелкие частицы полупереваренной пищи попадают внутрь эпителиально-мускульных пищеварительных клеток, где происходит внутриклеточное пищеварение. Непереваренные остатки выбрасываются через рот наружу.

Органы дыхания у кишечнополостных отсутствуют, а газообмен осуществляется через покровы тела.

Выделительная система. Продукты обмена (вода, диоксид углерода, мочевина, мочевая кислота, аммиак и др.) выделяются через эпителиальный слой эктодермы и энтодермы.

Размножение. Большинство кишечнополостных — раздельнополые животные, но есть и гермафродиты. У гидроидных половые продукты образуются в эктодерме, у остальных представителей их образование происходит в энтодерме. Оплодотворение у одних видов наружное (в воде), у других — внутреннее, в теле женских особей, куда проникают сперматии. Обычно развитие происходит со стадией личинки планулы, покрытой ресничками, позволяющими плануле плавать. У пресноводных гидр развитие прямое.

Тип Кишечнополостные делят на три класса: Гидроидные (*Hydrozoa*), Сцифоидные медузы (*Scyphozoa*) и Коралловые полипы (*Anthozoa*).

КЛАСС ГИДРОИДНЫЕ (*Hydrozoa*)

Низший класс кишечнополостных, состоящий примерно из 4 тыс. видов. Гидроидные представлены разнообразными одиночными и колониальными формами, населяющими преимущественно моря и океаны. Имеются и пресноводные представители. В отличие от сцифоидных медуз и коралловых полипов полипы и медузы, которые принадлежат к классу *Hydrozoa*, называются гидроидными. У гидроидных отсутствует глотка, стенки кишечной полости не имеют продольных перегородок. Половые продукты образуются в эктодерме.

Наиболее типичными для пресных вод являются различные виды гидр (*Hydra*), ведущие одиночный образ жизни полипа (рис. 36). Это небольшие животные высотой 1—2 см с расширенным основанием, на котором они удерживаются на субстрате. Ротовое отверстие окружено венчиком из 6—12 щупалец, а более широкое тело переходит в стебель. Мезоглэя имеет вид тонкой опорной пластинки, в которой разбросаны нервные, эпителиально-мускульные и промежуточные клетки. Из последних при необходимости формируются половые, стрекательные и другие клетки. Нервная система гидры имеет диффузный характер, хотя вокруг рта и на подошве находятся небольшие скопления нервных клеток. Эпителиально-мускульные клетки могут образовывать псевдоподии и поэтому способны к фагоцитозу.

Рис. 36. Пресноводная гидра *Hydra olidactis*:
1 — общий вид; 2 — продольный разрез; 3 — тело; 4 — подошва; 5 — щупальца; 6 — кишечная полость; 7 — эндодерма; 8 — эктодерма; 9 — опорная пластинка — мезоглея; 10 — семенники; 11 — образование яйца

Обитают гидры в пресных водоемах со стоячей или малоподвижной водой. Гидры могут медленно передвигаться за счет скольжения по-дошвы по субстрату или «кувырканием» через головной конец. Питаются мелкими ракообразными, инфузориями, коловратками и другими планктонными животными, улавливая добычу щупальцами, вооруженными стрекательными клетками.

Размножаются гидроидные почкованием и половым путем. При мерно на середине тела гидры имеется пояс почкования. Дочерние организмы отпочковываются и начинают самостоятельную жизнь в течение всего лета. Осенью гидры размножаются половым путем. На поверхности тела появляются особые выпуклости: несколько семенников или один-два яичника, в каждом из которых образуется только одна яйцеклетка. Гидры раздельнополы, но есть и гермафродиты. В последнем случае семенники на теле гидры образуются выше яичников. Спермин выходит в воду и проникают в яйцеклетку другой особи. Перекрестное оплодотворение у гермафродитных форм достигается разным временем созревания спермиев и яйцеклеток. Сначала развитие зиготы происходит в яичнике, затем зародыш покрывается оболочками, падает на дно и зимует. В таком состоянии зародыш может переносить промерзание и высыхание водоема. Весной из перезимовавшего зародыша вырастает гидра. Таким образом, у пресноводных гидр развитие прямое.

Гидры способны к регенерации, даже из части тела восстанавливается весь организм.

Среди обитателей морских вод подавляющее большинство гидроидных являются колониальными формами со сложным жизненным циклом (рис. 37). Колонии образуются путем многократного неполного почкования. В результате получается комплекс особей, сидящих на общем стволе и его побочных ветвях. Поэтому колония обычно напоминает бурые нарости мха или кустик, на ветвях которого сидят отдельные особи колонии — гидранты, похожие по строению на гидру. Кишечные полости всех гидрантов сообщаются между собой, т. е. пища и колонии может распределяться по всей колонии, что обеспечивает ее выживание. Для устойчивости и прочности за счет выделений эктодермального эпителия полипы образуют органическую оболочку — теку, одевающую не только общий ствол, но и отдельных гидрантов.

Размножение гидроидных полипов включает чередование бесполого поколения, ведущего прикрепленный образ жизни, и полового поколения — свободноплавающих гидроидных медуз (гидромедуз). В самих гидрантах колонии половые железы не образуются. Периодически на веточках колонии гидроидных полипов образуются особые почки,

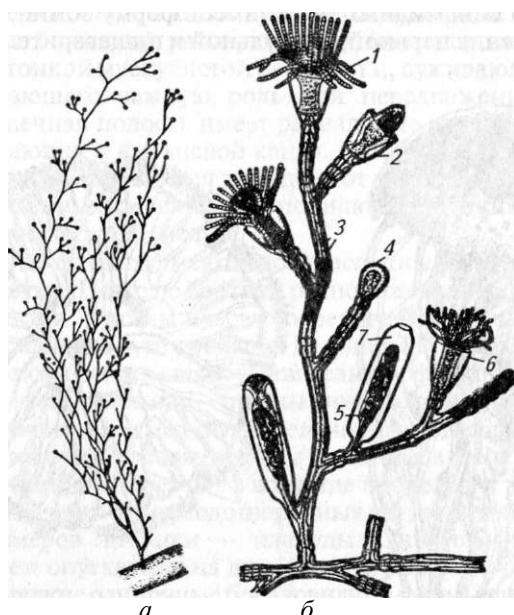


Рис. 37. Гидроид *Obelia*:
а — колония (слегка увеличено); б — отдельная веточка колонии (несколько схематизировано, часть особой колонии изображена в разрезе); 1 — гидрант в расправленном состоянии; 2 — сократившийся гидрант; 3 — тека; 4 — почка; 5 — бластостиль с развивающимися медузами; 6 — гидротека; 7 — гонотека (участок теки, одевающий бластостиль)

дающие начало половым особям — мелким гидроидным медузам. Эти медузы отрываются от материнской колонии и свободно плавают. Гидроидные медузы растут, и в них развиваются половые клетки. Медузы раздельнополы. Гидроидные медузы устроены значительно сложнее, чем гидроидные полипы; у медуз имеется нервное кольцо, статоцисты, глазки и т. п. Медузы ведут хищный образ жизни, захватывая и убивая щупальцами мелких животных, проглатывая и переваривая их в желудке. После созревания половые клетки выходят в воду и копулируют.

После копуляции гамет образуются личинки планулы, которые свободно плавают в воде с помощью многочисленных ресничек. Через некоторое время планулы опускаются на дно, прикрепляются к субстрату и превращаются в неподвижных полипов, которые дают начало новым колониям.

КЛАСС СЦИФОИДНЫЕ МЕДУЗЫ (Scyphozoa)

Класс, насчитывающий около 200 видов, представлен крупными и мелкими морскими медузами. Большая часть их жизненного цикла проходит в форме плавающих медуз (немногие формы ведут прикрепленный образ жизни); фаза полипа кратковременна или может отсутствовать. Тело сцифоидных медуз имеет форму зонта, купола и т. п. (рис. 38). Строение нервной, мускульной и пищеварительной систем у

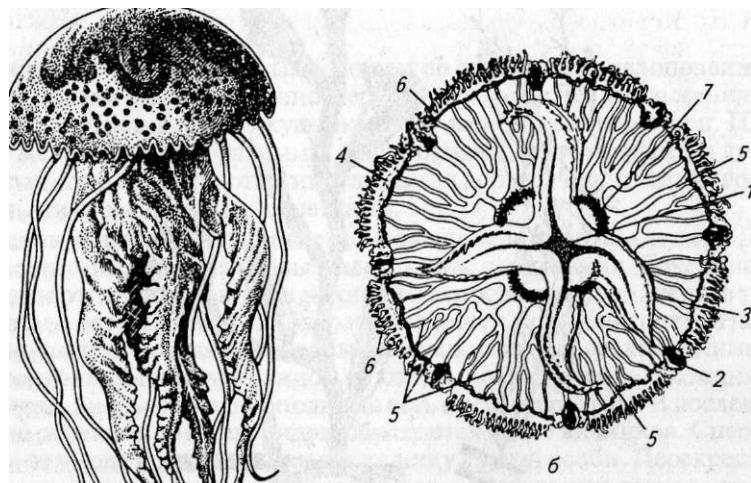


Рис. 38. Сцифоидные медузы:
а — медуза корнерот; б — схема строения аурелии; 7 — рот; 2 — ропалий; 3 — ротовые лопасти; 4 — кольцевой канал; 5 — радиальные каналы; 6 — щупальца; 7 — половые железы

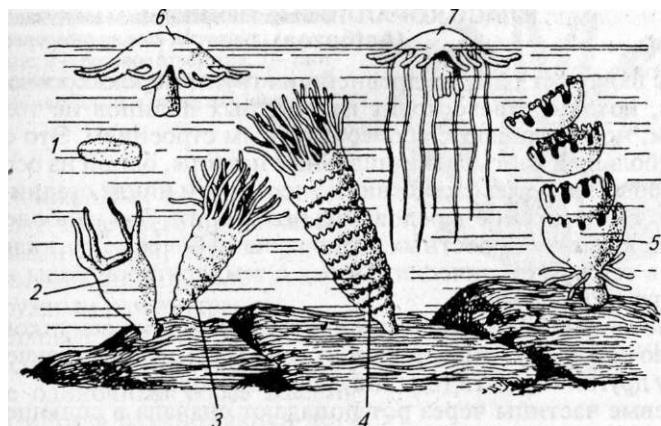


Рис. 39. Схема развития сцифоидной медузы аурелии (*Aurelia aurita*):
1 — личинка планула; 2 — полип сцифистомы; 3,4 — стадии почкования сцифистомы; 5 — отделение от сцифистомы личинок эфир; 6 — молодая медуза-эфира; 7 — взрослая медуза

этих медуз более сложное. В мезоглее купола имеются мышечные волокна, обеспечивающие сжатие купола. Сцифоидные медузы отличаются не только большими размерами тела, но и отсутствием специального паруса (тонкой мускулистой перепонки, суживающей кромки колокола), играющего важную роль при передвижении гидроидных медузок. Кишечная полость имеет радиальные складки и радиальные каналы, впадающие в кольцевой канал. Центральной частью пищеварительного аппарата является желудок, от которого отходит большое число разветвленных канальцев, выполняющих функции переноса питательных веществ в теле медуз.

Предротовые лопасти имеют многочисленные осязательные и стрекательные клетки. По краю зонтика расположены скопления нервных клеток — ганглии. Органы чувств сосредоточены в укороченных щупальцах — ропалиях. Внутри ропалии находится статоцист, а по бокам — два глазка, выполняющих светочувствительные функции. На щупальцах имеются обонятельные ямки — органы химического чувства.

В большинстве своем медузы раздельнополы. Половые продукты образуются в энтодерме: половые железы находятся в стенках желудка. Половые клетки выходят через рот в воду, где происходит копуляция мужских и женских гамет. Из оплодотворенных яиц развиваются микроскопических размеров личинки — планулы. Они плавают с помощью ресничек, затем опускаются на дно, прикрепляются к субстрату и превращаются в мелкие одиночные бокаловидной формы полипы — сцифистомы. По мере роста сцифистомы на ее теле появляются поперечные перетяжки, деля полип на ряд дисков — медуз (эфиры). Каждая эфира отделяется от сцифистомы, растет и превращается в свободноплавающую взрослую медузу. Таким образом, развитие сцифоидных медуз не прямое, а происходит через стадии планулы и сцифистомы (рис. 39).

КЛАСС КОРАЛЛОВЫЕ ПОЛИПЫ (Anthozoa)

Класс включает одну из древнейших групп морских животных — полипов, которые превосходят гидроидных полипов не только по размерам, но и отличаются более сложным строением. Это одиночные или большей частью колониальные полипы, одной из особенностей которых является отсутствие в жизненном цикле стадии медузы (рис. 40), т. е. у них нет чередования поколений. Это наиболее крупный класс кишечнополостных, включающий более 6 тыс. видов, обитающих в теплых тропических морях с температурой воды не ниже 20 °C на глубинах до 50 м.

Ротовое отверстие коралловых полипов окружено венчиком щупалец, число которых у одних полипов равно восьми (восьмилучевые кораллы), у других — шести (шестилучевые кораллы).

Пищевые частицы через рот попадают сначала в сплющенную с боков эктодермальную глотку, а оттуда — в хорошо развитую с перегородками (септами) кишечную полость. Число перегородок может быть либо восемь, либо шесть, или кратно шести — по числу щупалец. В глотке есть клетки с длинными ресничками, которые непрерывно гонят воду внутрь гастральной полости полипа, откуда вода выводится наружу. Так обеспечивается постоянная смена воды. Септы образованы мезоглеем, выстланной энтодермой (рис. 41). В нижней части полипа септы прикреплены только к стенке тела, в результате чего центральная часть гастральной полости (желудок) остается неразделенной.

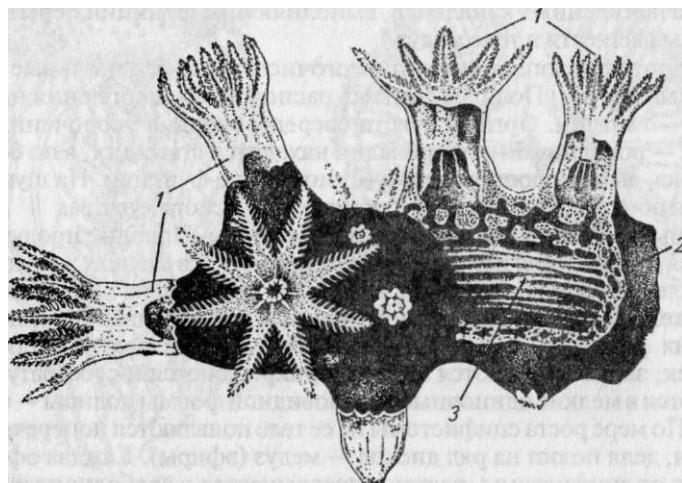


Рис. 40. Ветка колонии красного коралла:
1 — полипы; 2 — кора ветки; 3 — осевой скелет

Рис. 41. Схематическое изображение строения кораллового полипа *Alcyonium*:
1 — щупальце; 2 — ротовое отверстие; 3 — глотка; 4 — перегородки кишечной полости — септы; 5 — мезентериальные нити; 6 — яйца

У колониальных полипов мощный скелет чаще всего представлен углекислыми солями, реже — рогоподобным веществом. Скелет может быть наружным или внутренним.

Коралловые полипы размножаются бесполым и половым путем. Ведущие одиночный образ жизни актинии иногда размножаются делением, у колониальных видов наблюдается почкование. Половые железы формируются в перегородках между энтодермой и мезоглеей. Спермин выходят через ротовое отверстие наружу и через рот же проникают в гастральную полость женской особи, где и происходит оплодотворение. У некоторых форм оплодотворение наружное. Развитие происходит с метаморфозом: из зиготы развивается плавающая личинка — планула, которая прикрепляется к субстрату и дает начало новому полипу.

Актинии — одиночные шестилучевые яркоокрашенные полипы, лишенные скелета (рис. 42). Они могут медленно передвигаться с помощью мускулистой подошвы. Актинии очень чувствительны к раздражениям, сильно сокращаются, превращаясь в небольшой комок. Это хищники, питающиеся ракообразными, моллюсками и другими крупными животными, которых они захватывают щупальцами, парализуя стрекательными нитями. Некоторые актинии живут в симбиозе с раками-отшельниками, поселяясь на их раковинах. Рак служит для актиний средством передвижения, а актинии пассивно защищают рака от хищников.

В тропиках распространены рифообразующие мадрепоровые шес-

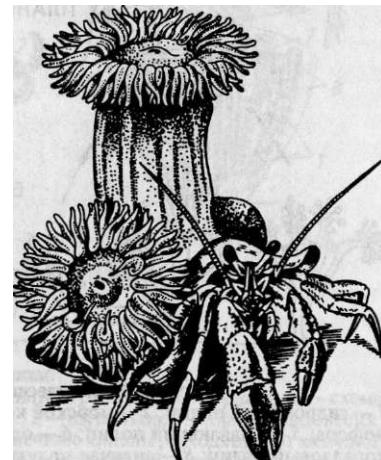
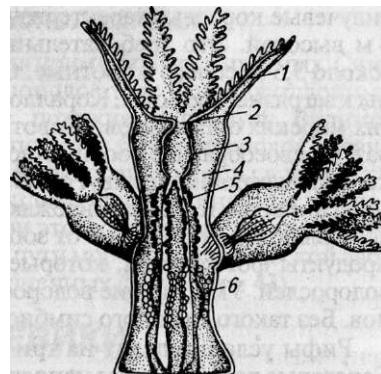


Рис. 42. Актиния на раковине, занятой раком-отшельником

тилучевые кораллы, характеризующиеся крупными размерами — более 4 м высотой. Это требовательные к температуре воды и ее солености (около 3,5 % солей) животные. Очень чувствительны коралловые полипы к загрязнению воды. Коралловые рифы служат местом обитания многих морских организмов. Животные и растения в коралловых рифах образуют своеобразное сообщество (биоценоз) рифа. В клетках энтодермы коралловых полипов живут симбиотические одноклеточные водоросли — зооксантеллы. Кораллы снабжают водоросли диоксидом углерода и предоставляют им укрытие, а от зооксантелл кораллы получают кислород и продукты фотосинтеза, которые поступают непосредственно из клеток водорослей. Умирающие водоросли перевариваются в цитоплазме полипов. Без такого сложного симбиоза коралловые полипы погибают.

Рифы условно делят на три типа: береговые, барьерные и атоллы. Береговые расположены непосредственно по берегам островов или материков, барьерные рифы располагаются параллельно береговой линии на некотором расстоянии. Атоллы — это кольцеобразные возвышающиеся над океаном коралловые острова с озерцом внутри.

Значение кишечнополостных в Мировом океане трудно переоценить: с их помощью осуществляется круговорот кальция в биосфере, они очищают морскую воду от органической взвеси, являются звеньями в пищевых цепях и т. п. Кишечнополостные служат объектом промысла: медузы (Япония, Китай), кораллы для украшений, коллекций и ювелирных изделий, медицинских препаратов.

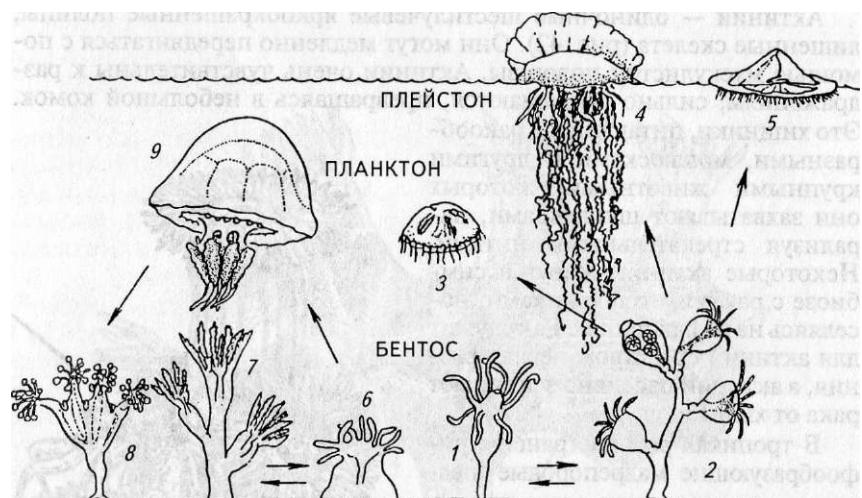


Рис. 43. Экологическая радиация кишечнополостных:
1 — гидроидный полип; 2 — морской колониальный полип; 3 — гидромедуза; 4 — сифонофора; 5 — плавающий полип; 6 — одиночный коралловый полип; 7 — колониальный коралловый полип; 8 — сидячая медуза; 9 — сцифоидная медуза

ФИЛОГЕНИЯ КИШЕЧНОПОЛОСТНЫХ

Кишечнополостные — древняя группа примитивных животных. Считается, что далекими предками кишечнополостных были двухслойные плавающие многоклеточные животные, похожие на планулу. Видимо, первыми были одиночные полипы. От полипов без перегородок развивались различные группы гидроидных. Коралловые полипы в процессе эволюции дали широкий спектр полипоидных форм: одиночных и колониальных, со скелетом и без него, но при этом сохранили древний признак развития без метагенеза. Основные пути морфо-экологической эволюции отражены на схеме радиации жизненных форм (рис. 43).

ТИП ГРЕБНЕВИКИ (*Ctenophora*)

Гребневики — это морские животные, характеризующиеся радиальной симметрией; они ведут одиночный свободноплавающий образ жизни, реже встречаются ползающие или сидячие формы. Известно около 120 видов гребневиков, заселяющих все моря. Питаются эти животные обычно планктоном.

Форма прозрачного и нежного тела мешковидная или грушевидная (рис. 44). Вдоль тела тянутся восемь рядов тонких и прозрачных гребных пластинок, образованных сросшимися ресничками. Пластинки

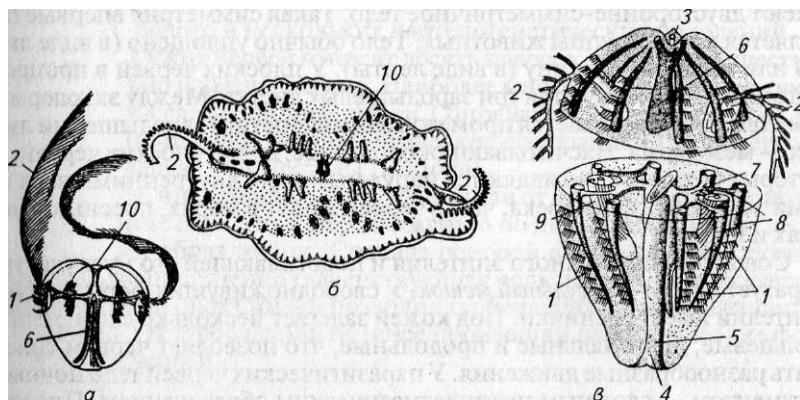


Рис. 44. Гребневики:
a — плавающий гребневик *Bolinopsis*; *б* — ползающий гребневик *Coeloplana*; *в* — схема строения гребневика; 1 — ряды гребных пластинок; 2 — щупальца; 3 — орган равновесия; 4 — рот; 5 — глотка; 6 — кишечная полость; 7 — кишечные каналы, отходящие от желудка; 8 — продольные кишечные каналы; 9 — влагалище щупальца; 10 — органы чувств

расщеплены наподобие гребешка и бьют по воде в одном направлении, что позволяет гребневикам двигаться только в одном направлении — ротовым концом тела вперед. Многие гребневики имеют пару щупалец с расположенными на них особыми клетками, вырабатывающими липкий секрет. С помощью этих щупалец гребневики захватывают разных животных, которыми питаются.

Рот, расположенный на оральном конце тела, ведет в глотку, которая переходит в короткий пищевод и затем в кишечную полость. Кишечная полость имеет разветвленные каналы и слепые отростки. Это двухслойные животные: снаружи тело покрыто эктодермой, а кишечная полость и ее каналы выстланы энтодермальным эпителием. Между экто- и энтодермой лежит студенистая прозрачная мезоглея. Наличие в мезогле многочисленных клеточных элементов и отсутствие стрекательных клеток отличает гребневиков от кишечнополостных.

Гребневики — гермафродиты; половые железы расположены в стенах пищеварительных каналов. Половые клетки выходят в просвет кишечных каналов и оттуда через рот наружу. Оплодотворение наружное. В развивающемся эмбрионе происходит образование зачаточного третьего зародышевого листка — мезодермы. Развитие без метаморфоза.

ТИП ПЛОСКИЕ ЧЕРВИ (*Plathelminthes*)

Общая характеристика. Большинство представителей этого типа имеют двусторонне-симметричное тело. Такая симметрия впервые появляется у этой группы животных. Тело обычно уплощено (в виде листа) или вытянуто в длину (в виде ленты). У плоских червей в процессе онтогенеза формируются три зародышевых листка. Между эктодермой и энтодермой развивается промежуточный (третий) зародышевый листок — мезодерма. Насчитывают около 15 тыс. видов плоских червей, из которых большинство являются наружными или внутренними паразитами животных и человека, часть червей живут в морях, пресных водоемах и почве.

Совокупность кожного эпителия и подстилающей его мускулатуры образует *кожно-мускульный мешок*. У свободноживущих форм кожный эпителий имеет реснички. Под кожей залегает несколько слоев мышц: кольцевые, диагональные и продольные, что позволяет червям совершать разнообразные движения. У паразитических червей тело покрыто тегументом — сложным цитоплазматическим образованием. Плоские черви не имеют полости тела (бесполостные), поскольку все пространство между внутренними органами и стенкой тела заполнено рыхло расположеннымными клетками мезодермального происхождения — паренхимой (паренхиматозные черви); в промежутках между клетками паренхимы циркулирует межтканевая жидкость. Паренхима выполняет опорные функции, в ней накапливаются резервные питательные вещества, она участвует в процессах обмена веществ.

Пищеварительный канал примитивен, обычно разветвлен и представлен двумя отделами: эктодермальной глоткой (передняя кишечник) и энтодермальной средней кишкой, которая заканчивается слепо. Задней кишки и анального отверстия нет. У части паразитических форм кишечник отсутствует.

Нервная система представлена парным мозговым ганглием и отходящими от него несколькими парами нервных стволов, идущих назад и соединенных между собой кольцевыми перемычками — комиссурами. Таким образом, у плоских червей формируется центральный аппарат 11 нервной системы. Органы чувств наиболее развиты у свободноживущих видов: имеются глазки, органы равновесия — статоцисты и многочисленные сенсиллы (осознательные клетки и органы химического чувства).

Кровеносная и дыхательная системы отсутствуют. Свободноживущие плоские черви дышат через кожу; для эндопаразитических форм характерно анаэробное дыхание.

У плоских червей появляются *органы выделения*, построенные по типу протонефридиев в виде системы разветвленных канальцев, оканчивающихся в паренхиме звездчатыми клетками с пучком ресничек внутри. Реснички способствуют откачке конечных продуктов обмена из паренхимы в один или два магистральных канала и затем через специальные выделительные отверстия (экскреторные поры) выводу этих продуктов наружу.

Плоские черви в большинстве своем гермафродиты. Половая система устроена сложно и обеспечивает внутреннее оплодотворение и высокую плодовитость. Развитие может быть прямым или с метаморфозом. Эндопаразитам присущи сложные жизненные циклы с чередованием обоеполого и нескольких партеногенетических поколений.

К типу плоских червей относят десять классов, из которых шесть — исключительно паразитические. Наиболее многочисленными являются четыре класса: Ресничные черви (*Turbellaria*), Дигенетические сосальщики (*Trematoda*), Моногенетические сосальщики (*Monogenea*) и Ленточные черви, или Цестоды (*Cestoda*).

Считается, что плоские черви произошли от древних кишечнополостных, которые перешли к передвижению по дну, где они могли вести хищнический образ жизни. Сначала основой для движения служили I-спички, но постепенно главенствующая роль перешла к мускулатуре гена. Активный образ жизни позволил турбелляриям усложнить систему органов. От турбеллярий позднее произошли паразитические формы плоских червей.

КЛАСС РЕСНИЧНЫЕ ЧЕРВИ (*Turbellaria*)

К классу ресничных червей относится большая группа (около 10 000 видов) свободноживущих в воде или в почве плоских червей, гибкость которых не расчленено и покрыто мерцательным (ресничным) мштелием. Все турбеллярии — хищники. На переднем конце тела ресничных червей имеется несколько примитивных глазков. У большин-

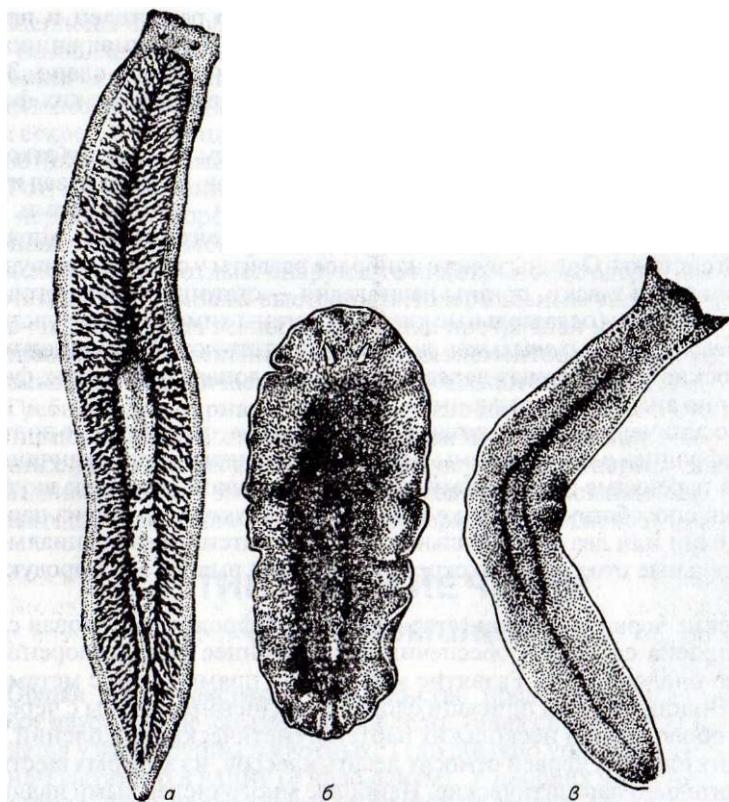


Рис. 45. Виды ресничных червей:
а — молочная планария *Dendrocoelum lacteum*; б — морская турбеллярия *Leptoplana*;
в — планария многоглазка *Polycoelis cornuta*

ства представителей рот расположен посередине тела на его брюшной стороне.

Строение и жизненные отравления. Длина тела может колебаться от долей миллиметра до 35 см. Форма тела уплощена и чрезвычайно разнообразна. Морские турбеллярии ярко окрашены (рис. 45).

Покровы представлены ресничным однослойным эпителием. Реснички способствуют передвижению мелких червей в воде, а более крупные представители ползают, вытягивая, сокращая и изгибая тело. В кожном эпителии располагаются особые палочковидные образования — рабдиты, выполняющие защитные функции: выбрасываясь наружу, они окутывают врага рыхлой клейкой оболочкой. В покровах ресничных червей много железистых клеток, одни из которых выделяют слизь, а другие — ядовитые вещества. Так, молочная планария на-

крыывает жертву своим телом и убивает ее ядом, который вырабатывают ядовитые железы, находящиеся на брюшной стороне тела червя.

Нервная система у разных представителей различна по своей сложности. У примитивных форм она диффузного типа. Есть виды, у которых вдоль тела идет несколько нервных тяжей. У более сложно организованных имеются ганглии с продольными нервными тяжами. Органы чувств представлены примитивными глазками, статоцистами и осознательными клетками.

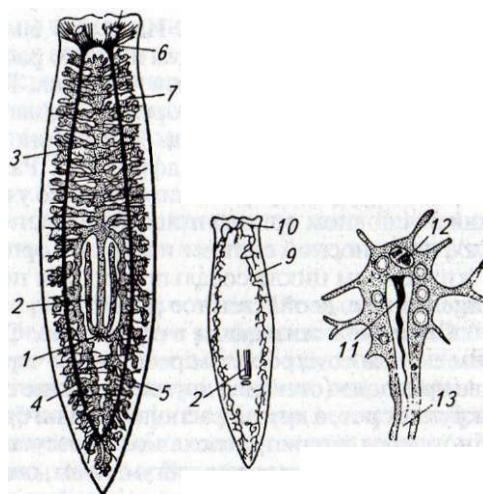
Органы пищеварения. У большинства видов на брюшной стороне в средней ее части расположен рот, ведущий в глотку, которая может выпячиваться наружу, присасываться к жертве и высасывать ее содержимое. Кишечник чаще всего ветвится на две и более ветвей, заканчивающихся слепо. Пища переваривается в полости кишечника и с помощью фагоцитоза (внутриклеточно) в клетках эпителия кишечной полости. Непереваренные остатки пищи выбрасываются наружу через рот. У некоторых ресничных червей кишечник отсутствует, а пища переваривается в пищеварительных вакуолях клеток, расположенных в специально обособленной пищеварительной паренхиме.

Газообмен осуществляется путем диффузии кислорода из воды через покровы внутрь тела, а диоксида углерода — наружу.

Выделительная система протонефридиального типа (рис. 46) как отдельная система органов впервые появляется у ресничных червей. У многих морских червей органов выделения нет: продукты обмена удаляются из тела через покровы и стенки кишечника.

Органы размножения устроены сложно. Большая часть ресничных червей гермафродиты. Мужская половая система представлена множе-

Рис. 46. Нервная система, органы пищеварения и выделения планарии:
а — пищеварительная и нервная системы; б — расположение главных канальцев выделительной системы; в — одна из концевых клеток протонефридиальной системы; / — рот; 2 — глотка; 3 — передняя ветвь кишечника; 4,5 — задние ветви кишечника; 6 — головной нервный узел; 7 — боковой нервный ствол; 8 — глазок; 9 — канальцы выделительной системы; К — выделительная пора; 11 — «мерцательное пламя»; 12 — ядро клетки; 13 — внутриклеточный каналец



ством мелких семенников, разбросанных в паренхиме. От семенников отходят семявыносящие каналы, которые, сливаясь, образуют два семяпроводы. Семяпроводы формируют непарный семязвергательный канал, пронизывающий совокупительный орган, расположенный в половой клоаке. Сюда же впадают и женские половые протоки.

Женская половая система представлена одним или множеством яичников. От яичников отходят два яйцевода, принимающие протоки желточников и сливающиеся в один канал — влагалище. Влагалище открывается в половую клоаку. Оплодотворенные яйцеклетки окружаются желточными клетками и вместе с ними покрываются общей скорлупой.

Благодаря разным срокам созревания половых продуктов самооплодотворения у этих червей не происходит. Оплодотворение внутреннее. Оплодотворенные яйца выводятся наружу либо через разрывы стенок тела, либо через рот, либо через специальные выводные протоки. У пресноводных форм развитие прямое. У морских видов развитие с превращением: из яйца развивается планктонная личинка, плавающая с помощью ресничек в толще воды.

Интересна способность ресничных червей к регенерации: при расчленении одного червя на сотни частей из каждой части может восстановиться новая особь.

В морях и океанах обитают мелкие ресничные черви из отрядов Бескишечные, Макростомиды, Многоветвистокишечные и др. Среди пресноводных представителей отечественной фауны можно отметить молочно-белую планарию, многочисленные трехветвистокишечные турбеллярии населяют озеро Байкал. Многие виды ресничных червей служат кормом для рыб.

КЛАСС СОСАЛЬЩИКИ (*Trematoda*)

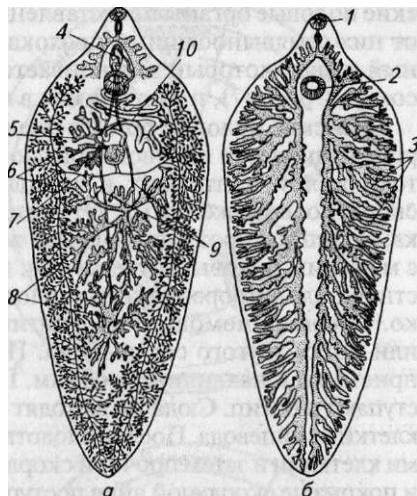
Общая характеристика. Известно более 4 тыс. видов сосальщиков, многие из которых являются широко распространенными и опасными эндопаразитами животных и человека. Тело сосальщиков не расчленено и имеет листовидную форму. У большинства есть присоски для прикрепления к телу хозяина. Кишечник двутрубный, заканчивается слепо. Большинство гермафродиты. Развитие происходит со сменой хозяев. Взрослые формы живут только у позвоночных животных, поражая в основном органы пищеварения, но есть виды, обитающие в легких, кровеносной системе и в других органах. В связи со сменой хозяев в жизненном цикле сосальщиков они получили еще одно название — *Digenea*, т. е. двойственное развитие.

Строение и жизненные отправления. Строение сосальщиков во многом сходно со строением ресничных червей. На теле обычно имеются две присоски (отсюда и другое название сосальщиков — двуустки): одна окружает рот, а другая расположена на брюшной стороне тела (рис. 47). Иногда брюшная присоска может отсутствовать.

Покровы представлены тегументом, снаружи у некоторых форм усеянным шипиками, способствующими фиксации паразита в теле хозяина.

Рис. 47. Строение половой (л) и пищеварительной (б) систем печеночного сосальщика фасциолы:

1 — ротовая присоска; 2 — брюшная присоска; 3 — разветвленный кишечник (левая и правая ветви); 4 — копулятивный орган; 5 — яичник; 6 — желточники; 7 — желточные протоки; 8 — семенники; 9 — семяпроводы; 10 — матка



Мускулатура представлена слоями мышечных волокон, которые вместе с тегументом образуют кожно-мускульный мешок. Двигаются трематоды медленно.

Нервная система слагается из парного головного ганглия и отходящих от него парных нервных тяжей. От ганглия и тяжей идут ответвления ко всем органам. У взрослых червей органы зрения отсутствуют. В покровах размещены осязательные и другие нервные окончания.

Органы пищеварения начинаются ротовым отверстием, ведущим в глотку, которая может совершать сосательные движения. За глоткой лежит пищевод, который ветвится на две ветви кишечника. Иногда ветвистый кишечник имеет дополнительные боковые отростки, облегчающие распределение продуктов пищеварения в паренхиме червя. Анальное отверстие отсутствует

Пищеварение в основном происходит в полости кишечника (внеклеточное). Наряду с кишечным пищеварением наблюдается всасывание растворенных органических веществ через покровы тела. Непереваренные остатки пищи выбрасываются через ротовое отверстие. Продукты пищеварения транспортируются в теле сосальщиков с помощью межтканевой жидкости и клеток паренхимы за счет сокращения мускулатуры тела.

Органы дыхания отсутствуют. У эндопаразитических форм дыхание анаэробное, процессы диссимиляции происходят по типу брожения. Но есть и исключения: паразиты, живущие в легких, могут дышать через покровы своего тела.

Органы выделения протонефридиального типа. От выделительных звездчатых клеток, покрытых ресничками, и разбросанных в паренхиме, отходят канальцы, которые сливаются в более крупные выделительные каналы. Вся выделительная система каналов открывается в мочевой пузырь, а из него конечные продукты обмена веществ выбираются через выделительное отверстие на заднем конце тела.

Органы размножения устроены сложно. Все трематоды гермафродиты, лишь немногие виды кровяных сосальщиков раздельнополы. Муж-

ские половые органы представлены двумя семенниками и отходящими от них семявыносящими протоками, образующими семязвергательный канал, который заканчивается копулятивным органом — циррусом (см. рис. 47), находящимся в половой клоаке.

Женские половые органы представлены одним яичником, от которого начинается яйцевод, впадающий в оотип. Сюда же впадают протоки желез: желточных и тельца Мелиса. Оотип окружен мелкими скорлуповыми железами. От оотипа начинается длинная извитая матка, конец которой открывается женским половым отверстием рядом с мужским половым отверстием. Сосальщикам свойственно перекрестное оплодотворение, самооплодотворение возможно, но очень редко. Сперма с помощью копулятивного органа вводится в свою матку или матку другого сосальщика. По матке спермии поступают в семяприемник и накапливаются там. По мере необходимости спермии поступают в оотип. Сюда же выходят продукты придаточных желез и яйцеклетки из яйцевода. После оплодотворения яйца покрываются желточными клетками и затем прочной скорлуповой оболочкой. Оплодотворенные и покрытые скорлупой яйца поступают в матку и выводятся наружу. Яйцо имеет крышечку, открывающуюся при выходе из него личинки.

Развитие большинства сосальщиков протекает со сложными превращениями и со сменой хозяев. Первые промежуточные хозяева — всегда брюхоногие моллюски, пресноводные или наземные. Вторые промежуточные (дополнительные) хозяева (если они есть) — разные беспозвоночные и позвоночные животные. Есть виды сосальщиков, которые имеют трех промежуточных хозяев (рис. 48).

Наиболее опасны как паразиты сельскохозяйственных животных и человека следующие дигенетические сосальщики.

Печеночный сосальщик (*Fasciola hepatica*), или фасциола печеночная, имеет листовидное тело до 5 см длиной (см. рис. 47). На переднем конце тела расположено ротовое отверстие, окруженное ротовой присоской. Тегумент с шипиками. На брюшной стороне тела имеется брюшная присоска. Кишечник двутрубистый с множеством отростков. Два ветвистых семенника расположены в середине тела ниже компактного ветвистого яичника. По бокам тела находятся желточники.

Печеночный сосальщик паразитирует в желчных протоках печени растительноядных и всеядных животных, может поражать и человека, вызывая заболевание фасциолез. Питается сосальщик желчью. Сильнее всего поражаются овцы и молочный скот, которых пасут в поймах рек, на заливных лугах и т. п. Нередко болезнь может иметь летальный исход из-за закупоривания двустками желчных протоков и невозможности оттока желчи из печени.

В желчных протоках паразиты копулируют, но возможно и самооплодотворение. С желчью оплодотворенные яйца, покрытые скорлуповыми оболочками, через кишечный тракт с калом хозяина попадают во внешнюю среду. За сутки один паразит может отложить сотни тысяч яиц (рис. 49).

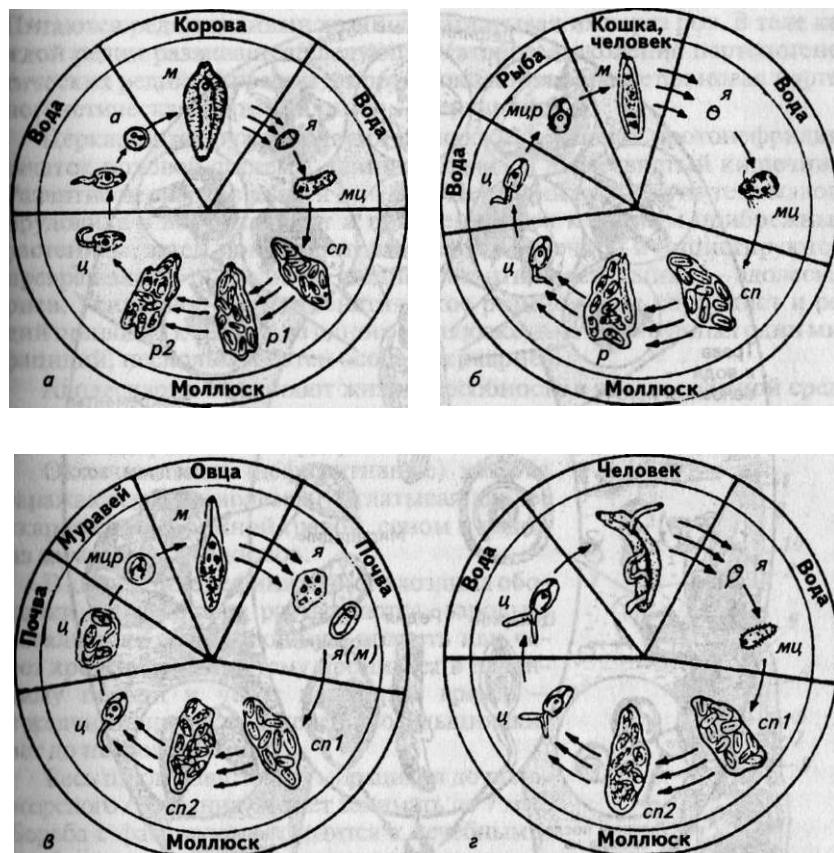


Рис. 48. Схемы жизненных циклов третратод:
 а — печеночный сосальщик; б — кошачья двуустка; в — ланцетовидный сосальщик;
 г — кровяной сосальщик; м — марита; я — яйцо; миц — мирадицид; сп — спороциста;
 р — редия; и — церкарий; мир — метацеркарий

Для дальнейшего развития яйцо должно попасть в воду, где через 3—6 нед крышечка открывается и из яйца выходит микроскопическая личинка, покрытая ресничками, — мирадицид. На переднем конце мирадиции имеются глазки и рот, ведущий в кишечник. Свободно плавать мирадиции могут не более 2 сут. Они не питаются. Если мирадиции не попадут в тело промежуточного хозяина, то они погибают. В задней части тела мирадиции лежат партеногенетические яйца. Найдя промежуточного хозяина (пресноводный брюхоногий моллюск малый прудовик, *Limnaea truncatula*), мирадицид с помощью специальных железок, расположенных около рта, проникает в моллюска и превращается в следующую стадию развития — спороцисту. Это половозрелая ста-

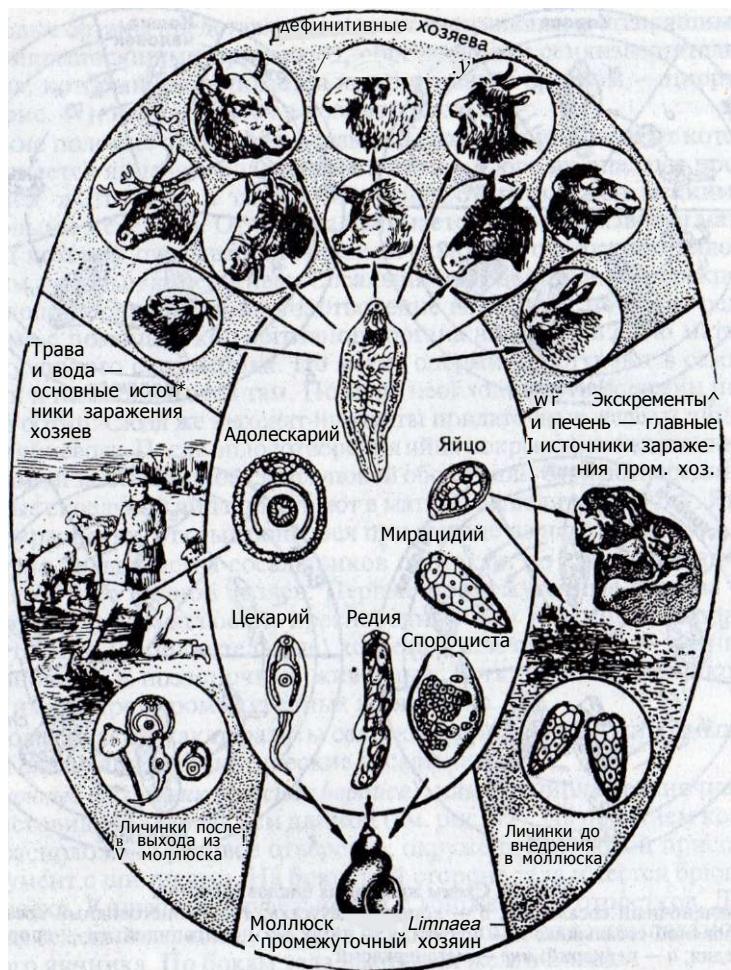


Рис. 49. Цикл развития печеночного сосальщика фасциолы (*Fasciola hepatica*)

дия, которая способна к размножению. Иными словами, мириацидий является как бы личинкой спороцисты.

Спороциста лишена ресничек, она выделяет в ткани хозяина пищеварительные ферменты и питается продуктами гидролиза тканей хозяина, поглощая эти продукты всей поверхностью тела. Внутри спороцисты путем дробления и дифференцировки в течение 3—4 нед из партеногенетических яиц развивается новое поколение личинок — редии. Редии выходят из погибшей спороцисты через разрывы ее стенок. Редии имеют вытянутое тело с развитым пищеварительным аппаратом.

11итаются редии тканями хозяина, заглатывая их через рот. В теле каждой редии развивается следующее (второе) поколение партеногенетических редий. Из редий второго поколения образуется новая партеногенетическая форма личинок — церкарии.

Церкарии вооружены двумя присосками, имеют протонефридии, зачаток половой системы, длинный хвост и двуветвистый кишечник. Развитие церкариев длится до 6 нед. Церкарии выходят из тела малого прудовика в воду, плавают и прикрепляются к водным прибрежным растениям, затем покрываются плотной оболочкой и инцистируются, превращаясь в последнюю стадию развития сосальщика —adolескариев. Усиленное партеногенетическое размножение спороцист и редий приводит к выходу из одного моллюска, в которого попал один мириацидий, нескольких сотен особей церкариев.

Адолескарии сохраняют жизнеспособность в воде и влажной среде многие месяцы, долго живут они в сене, заготовленном из растений, на которых закрепились церкарии.

Окончательные (дефинитивные) хозяева заражаются фасциолезом, заглатывая адолоскариев с прибрежной травой, сеном и водой из зараженных водоемов.

В кишечнике дефинитивного хозяина оболочки адолоскариев растворяются, зародыш сосальщика через брюшную полость или через кровеносную систему проникает в паренхиму печени и через некоторое время — и желчные протоки. В печени сосальщик живет до нескольких лет.

Весь цикл развития (от мириацидия до половозрелого состояния) может занимать до 7 мес. Борьба с фасциолезом сводится к лечебным и профилактическим мерам с целью предупредить попадание адолоскариев в организм сельскохозяйственных животных (мелиорация пастбищ, поение скота чистой водой из специальных сооружений, смена пастбищ и т. д.).

Ланцетовидный сосальщик (*Dicrocoelium lanceatum*), или ланцетовидная двуустка, — небольшой червь (около 1 см) ланцетовидной формы, распространен в засушливых регионах страны. Имеет две присоски, двуветвистый кишечник без боковых отростков, яичник и семенники неветвящиеся (рис. 50). Паразитирует в печени мелкого и крупного рогатого скота и других травоядных млекопитающих.

Первым промежуточным хозяином лан-
I (стовидного сосальщика служат разные виды сухопутных брюхоногих моллюсков. С кало-

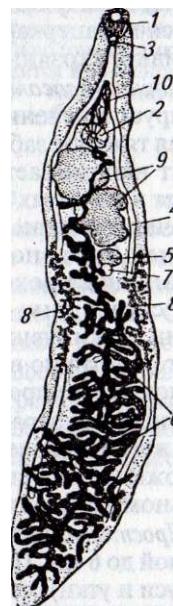


Рис. 50. Ланцетовидный сосальщик *Dicrocoelium lanceatum*:

1 — ротовая присоска; 2 — брюшная присоска; 3 — глотка; 4 — ветви кишечника; 5 — яичник; 6 — матка; 7 — семяприемник; 8 — желточники; 9 — семенники; 10 — семязвергательный канал

выми массами дефинитивного хозяина выходят яйца сосальщика, окруженные толстыми оболочками, которые позволяют сохранять жизнеспособность в течение нескольких месяцев. В яйцах находятся сформированные мирадии. Моллюски заражаются, поедая эти яйца. В кишечнике моллюсков мирадии освобождаются от яйцевых оболочек и проникают в печень промежуточного хозяина. Там мирадии превращаются в спороцисты, в которых партеногенетически развиваются дочерние спороцисты. Последние дают начало партеногенетическим церкариям. Сотни церкариев проникают в легкие моллюска, обволакиваются слизью в комочки (пакеты церкарий) и в таком виде выбрасываются через дыхательное отверстие наружу.

Вторым промежуточным хозяином являются разные виды муравьев. Муравьи могут поедать слизистые комочки с церкариями. Через стенку кишечника муравьев церкарии проникают в полость тела насекомых, где превращаются в метацеркариев (без хвоста, окруженные оболочками).

Скот, заглатывая траву с пораженными муравьями, заражается сосальщиком. В кишечнике скота муравьи перевариваются, а освободившиеся метацеркарии через желчный проток попадают в печень дефинитивного хозяина.

Кошачий сосальщик (*Opistorchisfelineus*), или кошачья двуустка, паразитирует в печени хищных млекопитающих, а иногда и человека, вызывая тяжелое заболевание — описторхоз. По строению и размерам паразит напоминает ланцетовидного сосальщика. Кошачий сосальщик живет в желчных протоках печени, в желчном пузыре и даже в поджелудочной железе. Первым промежуточным хозяином служит пресноводный брюхоногий моллюск битиния (*Bithynia leachi*), в котором мирадий проходит те же стадии цикла развития, что и печеночный сосальщик: мирадий—спороциста—редии—церкарии. Церкарии, покинувшие первого промежуточного хозяина, выходят в воду и затем активно внедряются в тело рыб, в основном различных видов карповых, где превращаются в метацеркариев.

Хищники заражаются, съедая рыбу, пораженную метацеркариями. Заражение человека и животных происходит при поедании вяленой, мороженой или сырой рыбы, инфицированной метацеркариями. При сильном заражении болезнь может закончиться смертью человека.

Простогонимусы (виды рода *Prosthogonimus*) — небольшие сосальщики, длиной до 6 мм. Паразитируют эти черви в яйцеводах птиц (куриные, реже гуси и утки), что приводит к образованию бесскорлупных яиц и к последующему прекращению яйцекладки. Промежуточными хозяевами являются различные виды пресноводных моллюсков, а дополнительными хозяевами — личинки стрекоз. Яйца развиваются в воде. Вышедшие из них мирадии проникают в тело моллюсков, в печени которых превращаются в спороцисты. Последние дают начало партеногенетическим церкариям. Выходя из тела моллюсков, церкарии с водой попадают в кишечник личинок стрекоз. Там они лишаются хвоста, проникают в разные части тела личинок и превращаются в метацеркариев. Поедая личинок и взрослых стрекоз, птицы заражаются простогонимусами.

Кровяные двуустки (несколько видов из рода *Schistosoma*) существенно отличаются от других групп трематод. Длина тела до 2 см. Они раздельнополы. Самка размещается в специальном желобе на брюхе самца (рис. 51). Паразитируют они в венах пищеварительной и выделительной систем птиц, млекопитающих и человека. Оплодотворенные яйца с помощью особого шипа проникают в заднюю часть кишечника и с калом (у птиц с пометом) выносятся наружу. В воде из яиц выходят мирандии и внедряются в пресноводных брюхоногих моллюсков. В моллюсках паразит проходит все стадии спороцисты, дочерней спороцисты и перкария. Из тела моллюска церкарии выходят в воду и через кожу проникают в дефинитивного хозяина, вызывая тяжелое заболевание — шистосомоз. В тропических странах шистосомозом поражено около 900 млн человек.

Для нормального прохождения всего ЦИК-ла развития сосальщикам требуются весьма благоприятные условия: наличие воды, промежуточных хозяев в ней, постоянное посещение водоемов дефинитивными хозяевами и т. д. Однако есть виды, отлично приспособившиеся к паразитированию. Например, сосальщик *Leucochloridium paradoxum* живет в кишечнике насекомоядных певчих птиц. В теле промежуточного хозяина — наземной улитки, съевшей яйца паразита, мирадии превращаются в спороцисты. Спороцисты разрастаются, образуя большое число отростков, распространяющихся по телу улитки. Некоторые отростки попадают в щупальца улитки, сильно раздуваются, приобретают яркую окраску, просвечивающую сквозь тонкую кожу улитки, и периодически сокращаются. Птицы склевывают эти щупальца, похожие на гусениц насекомых, заражаясь таким образом сосальщиком.

КЛАСС МОНОГЕНЕИ (*Monogenea*)

За редким исключением моногенетические сосальщики являются эктопаразитами позвоночных (рыб, амфибий) и беспозвоночных (моллюсков) животных. Известно более 2,5 тыс. видов моногеней. Паразитируют они на жабрах и коже рыб, некоторые поражают мочевой пузырь амфибий и рептилий. По своей организации моногеней близки к трематодам. Они обладают мощными органами для прикрепления к хозяину. Это присоски и крючья или только крючья, которые расположены на обособленном заднем отделе тела в виде диска, а также мелкие присоски около рта, выделяющие липкий секрет. Ротовая и брюш-

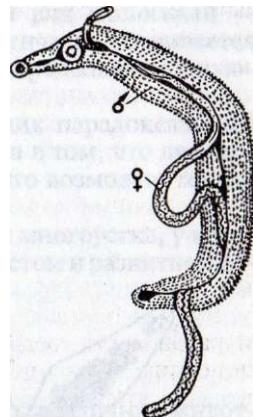


рис. 51. Кровяная двуустка
Schistosoma haematobium:
Г — лобке более широкого
самца (σ^{\wedge}) Т —

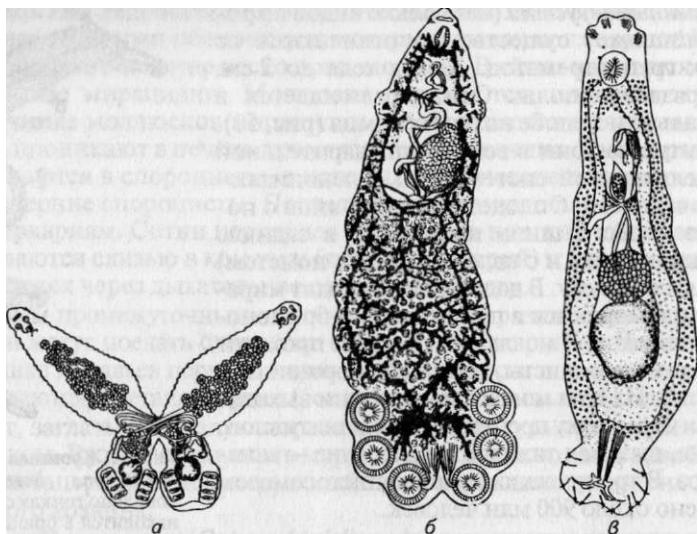


Рис. 52. Моногенетические сосальщики:
а — спайник парадоксальный *Diplozoon paradoxum*; б — лягушачья многоустка *Polystoma inlegerrimum*; в — дактилогирус *Dactylogyrus vastator*

ная присоски у моногеней отсутствуют (рис. 52). Из органов чувств можно отметить наличие на покровах многочисленных чувствующих клеток и на переднем конце — глазков.

Тело моногеней заключено в кожно-мускульный мешок. Кишечник двуветвистый, иногда с боковыми отростками или мешковидный. Половые железы непарные. Парные выделительные каналы протонефридиальной системы открываются на переднем конце тела парными отверстиями.

В половой системе имеется влагалище, по которому сперма вводится в оотип и семяприемник. Матка открывается самостоятельным отверстием в половую клоаку и служит только для выведения оплодотворенных яиц.

Размножаются моногеней исключительно половым путем, некоторым видам свойствен партеногенез. Жизненный цикл без смены хозяина, все развитие паразита проходит в одном хозяине.

Из оплодотворенного яйца выходит свободноплавающая личинка с ресничками, глазками и органами прикрепления на заднем конце тела. Личинка напоминает планарию.

Некоторые моногенетические сосальщики приносят вред рыбному хозяйству, поражая рыб и вызывая их истощение. Например, мелкие черви *Dactylogyrus vastator* длиной 1—3 мм живут на жабрах и коже карповых и других рыб, питаясь кровью. Из яиц, отложенных дактилогирусом, вылупляются личинки, которым затем надо прикрепиться к жабрам рыб, где они превращаются во взрослых паразитов.

На карпах часто паразитирует живородящий вид моногеней — *(Iyrodactylus elegans)*. В этом паразите партеногенетически развивается только один зародыш, в котором имеется еще три зародыша последующих поколений.

На жабрах карповых рыб паразитирует спайник парадоксальный (*Diplozoon paradoxum*). Особенность этого паразита в том, что две гермафродитные особи срастаются таким образом, что возможно только перекрестное оплодотворение.

Есть паразиты лягушек, в частности лягушачья многоустка, у которой жизненный цикл усложнен и тесно связан с ростом и развитием хозяев — головастиков и лягушек.

КЛАСС ЛЕНТОЧНЫЕ ЧЕРВИ (Cestoda)

Общая характеристика. Цестоды — эндопаразиты различных животных, преимущественно позвоночных, и человека. Взрослые черви паразитируют в тонком отделе кишечника дефинитивного хозяина; личинки паразитов развиваются в различных органах и полостях тела промежуточного хозяина — беспозвоночных и позвоночных животных. Известно более 3 тыс. видов цестод, среди которых много паразитов животных и человека.

У большинства представителей ленточных червей тело имеет вид Itosкой ленты, часто расчлененной на множество члеников. Тело имеет головку — сколекс, которая продолжается в шейку; за шейкой следует тело червя — стробила. Стробила состоит из множества (от нескольких тысяч до сотен, но бывает и два—четыре) члеников — пролоттид. Реже встречаются цестоды с нерасчлененным телом. Головка цестод имеет специальные органы прикрепления: присоски, крючья, ботрии (щелевидные углубления).

В связи с паразитическим образом жизни у ленточных червей слабо развиты нервная система и органы чувств, редуцирована пищеварительная система. Однако половая система достигает высокого уровня развития, обеспечивая огромную плодовитость, а следовательно, и возможность выживания паразитов.

Строение и жизненные отправления. Длина тела колеблется от нескольких миллиметров до 15 м. Головка (сколекс) имеет разное строение у различных цестод (рис. 53). У бычьего цепня сколекс

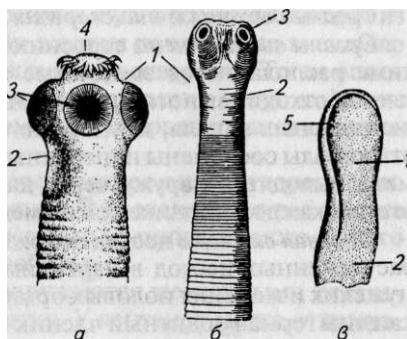


Рис. 53. Головки цепней и лентецов:
а — цепня вооруженного; б — цепня не вооруженного; в — лентца широкого;
1 — головка; 2 — шейка; 3 — присоски;
4 — хоботок с крючьями; 5 — присасывающие ямки — ботрии

леке имеет только четыре присоски (невооруженный цепень), головка свиного цепня помимо четырех присосок на вершине вооружена дополнительно венчиком из хитиновых крючьев (вооруженный цепень), головка широкого лентеца имеет два щелевидных углубления — ботрии, которыми они зажимают складку стенки кишечника хозяина. Еще более сложны органы прикрепления у гвоздичника и других цестод.

Шейка цестод является зоной роста черва, так как в ней происходит отшнуровывание новых членников, из которых состоит стробила. Только у немногих представителей (ремнец, гвоздичник) тело не подразделяется на членники. На заднем конце тела цестод находятся зрелые членники, наполненные яйцами червя. Они отрываются по мере созревания и увлекаются с калом хозяина во внешнюю среду. Таким образом, у цестод происходит постоянный прирост молодых членников и отрыв старых (зрелых) членников. Число членников у разных цестод может варьироваться в широких пределах: от двух—четырех до нескольких тысяч. В передней части стробилы расположены незрелые членники, у которых еще не развиты половые органы; за незрелыми следуют гермафродитные членники с развитой гермафродитной половой системой. Конец стробилы представлен зрелыми членниками с маткой, набитой яйцами.

Покровы представляют собой тегумент, который подстилают кольцевые и продольные слои мускулатуры. По сравнению с trematodами тегумент у цестод выполняет более многообразные функции: защита от действия пищеварительных ферментов кишечника хозяина путем их нейтрализации, всасывание питательных веществ из кишечного содержимого хозяина, выработка и выделение собственных ферментов и т. п. Тегумент имеет волоски и ворсинки, увеличивающие поверхность всасывания пищи.

Мускулатура представлена наружным кольцевым и внутренним продольным слоями. Может быть и третий — диагональный слой. В кишечнике хозяина ленточные черви совершают медленные движения. Такие же движения совершают и вышедшие наружу с калом зрелые членники.

Нервная система состоит из скопления в сколексе червя нервных клеток и продольных парных тяжей, идущих до конца тела. Органы чувств выражены слабо.

Органы дыхания и пищеварения у цестод отсутствуют.

Органы выделения по строению однотипны с trematodами. В паренхиме располагаются звездчатые клетки, несущие реснички; от этих клеток отходят выносящие канальцы, сливающиеся в два крупных выделительных канала, идущих по бокам стробилы. В каждом членнике эти каналы соединены поперечным протоком. Конечные продукты обмена выводятся наружу через каналы последнего членника. Помимо этого в каждом членнике тоже имеются отверстия протонефридиев.

Половая система цестод похожа наловую систему trematod. У нерасчлененных цестод в паренхиме расположен лишь один комплект мужских и женских половых органов (рис. 54). У расчлененных цестод каждый гермафродитный членник имеет по одному комплекту женских

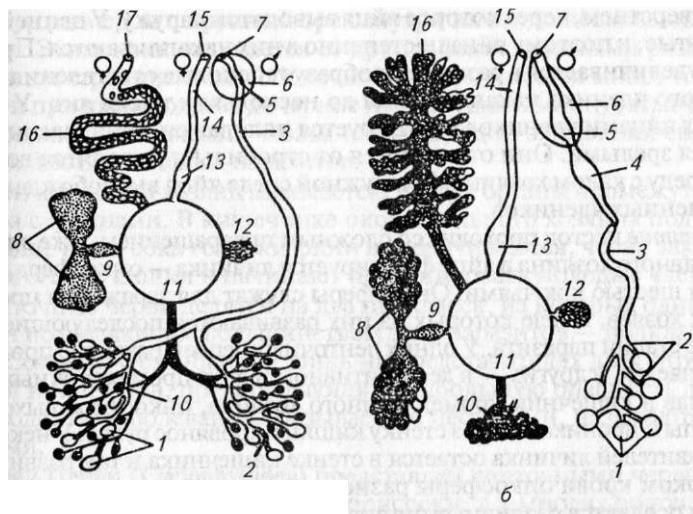


Рис. 54. Схема строения половой системы ленточных червей:
 ч — лентец (с открытой маткой); б — цепень (с закрытой маткой); 1 — семивыносящие каналы; 2 — семяпровод; 3, 4 — семяпровод; 5 — совокупительный орган; 6 — его сумка; 7 — мужское половое отверстие; 8 — яичники; 9 — яйцевод; 10 — желточник; II — оотип; 12 — тельце Мелиса; 13 — семяприемник; 14 — влагалище; 15 — женское половое отверстие; 16 — матка; 17 — отверстие матки

п одному комплекту мужских половых органов, а у некоторых представителей — по два комплекта. Мужская половая система представлена множеством мелких семенников, разбросанных в паренхиме. Отходящие от семенников семивыносящие каналы сливаются в семяпровод, который заканчивается семязвергательным каналом и копулятивным органом.

Женская половая система состоит из яйцевода, оотипа, матки, влагалища, семяприемника, желточников и тельца Мелиса. Влагалище и семязвергательный каналы открываются в половую клоаку. У одних ленточных червей матка открывается наружу специальным отверстием — открыта матка. В этом случае оплодотворенные яйца постоянно выходят из тела червя и вместе с каловыми массами хозяина попадают во внешнюю чаду. У других червей матка не имеет выводного отверстия. Тогда оплодотворенные яйца или развивающиеся в них личинки могут выйти из яйцеклетки дефинитивного хозяина только вместе с оторвавшимися от стробилы зрелыми членниками.

У ленточных червей происходит перекрестное оплодотворение, но может происходить и самооплодотворение. В последнем случае сперма может вводиться во влагалище собственного членика, а также любого другого гермафродитного членика стробилы.

Оплодотворенные в оотипе яйца окружаются желтовыми клетками и скорлупой, затем выводятся в матку. У лентецов матка открывается

ется отверстием, через которое яйца выводятся наружу. У цепней матки замкнутые, и поэтому яйца постепенно в них накапливаются. При этом матка увеличивается в размерах и образует боковые ответвления. В матке одного членика накапливается до нескольких тысяч яиц. У наполненных яйцами члеников редуцируется половая система, членики становятся зрелыми. Они отрываются от стробилы и выводятся во внешнюю среду с калом хозяина. В наружной среде яйца высвобождаются из разрушенных члеников.

Развитие цестод проходит со сложным превращением. Уже в теле дефинитивного хозяина в яйце формируется личинка — онкосфера, вооруженная шестью крючьями. Онкосфера служат для заражения промежуточных хозяев, в теле которых из них развиваются последующие личиночные стадии паразита. У одних ленточных червей развитие проходит в двух хозяевах, у других — в дефинитивном и двух промежуточных.

Попав в кишечник промежуточного хозяина, онкосфера выходит из скорлупы и проникает через стенку кишки в кровяное русло. У некоторых представителей личинка остается в стенке кишечника и там развивается.

С током крови онкосфера разносятся по телу промежуточного хозяина и оседают в различных органах и тканях, где образуют новую стадию личинки — финну. Строение финн у цепней неодинаково. Различают три типа финн: цистицерк, ценур и эхинококк. Цистицерк в виде небольшого округлого пузырька имеет одну впаянную внутрь головку будущего паразита (рис. 55). У лентецов встречается примитивная финна — плероцеркоид, имеющий лентовидную форму и одну ввернутую головку с ботриями.

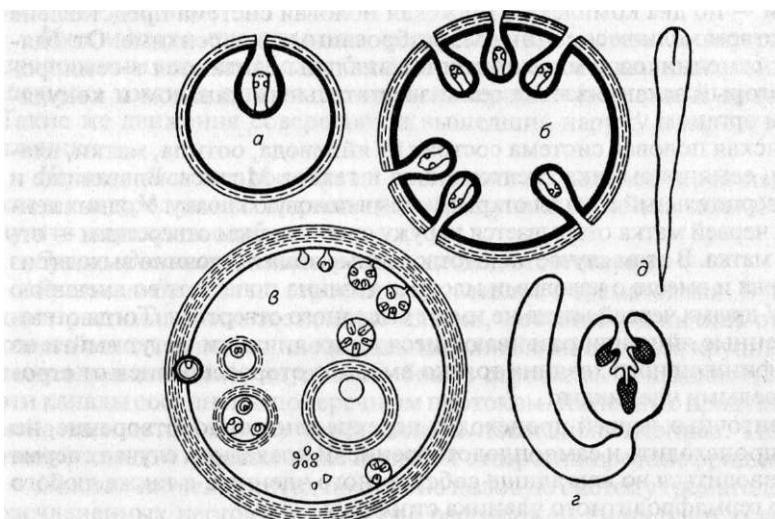


Рис. 55. Различные формы финн ленточных червей:
а — цистицерк; б — ценур; в — эхинококк; г — цистицеркоид; д — плероцеркоид

Ценур размером с крупный орех внутри имеет до нескольких сотен головок. Эхинококк отличается от других финн крупными размерами и сложным строением: под оболочкой пузыря имеется много головок, внутри пузыря находятся дочерние пузыри, дающие внучатые пузыри с несколькими головками. Таким образом, один эхинококк способен дать множество молодых паразитов.

Окончательный хозяин заражается, поедая органы промежуточного хозяина с финнами. В кишечнике окончательного хозяина под действием кишечного сока головки финн выворачиваются, черви закрепляются на стенке кишки и начинают продуцировать молодые членики.

Ленточные черви делятся на два подкласса, из которых один представлен подавляющим большинством этих паразитов — подкласс Цестоды (Cestoda). Остановимся лишь на представителях двух отрядов — I (епней и Лентецов, представляющих наибольшую опасность для животноводства. Заболевания, вызываемые цестодами, называются цестодозами.

Отряд Цепни (Cyclophyllidea) представлен ленточными червями, на скелете которых имеется четыре присоски, а у многих видов дополнительно есть венчики крючьев. Матки закрытого типа. Онкосфераe развиваются не во внешней среде (лентецы), а в матках зрелых члеников. Половая клоака находится сбоку членика.

Невооруженный (бычий) цепень (*Taeniarhynchus saginatus*) достигает в длину 8—12 м. На головке расположены только четыре присоски, а хоботка с крючьями нет. Паразитирует только в кишечнике человека. К матке зрелого членика может быть до 100 тыс. яиц; матка сильно разветвлена — до 35 ответвлений с каждой стороны (рис. 56).

Вышедшие с калом человека зрелые членики могут передвигаться. Промежуточный хозяин — крупный рогатый скот, заражается, проглатывая яйца с кормом и водой. В кишечнике скота из яиц выходят онкосфераe, которые вбуравливаются в стенку кишечника и проникают в кровь. Онкосфераe оседают в мышцах внутренних органов, где образуются финны типа цистицерк. Человек может заразиться, потребляя плохо проваренное или недожаренное мясо пораженного скота.

Заболевший человек худеет, происходит интоксикация организма продуктами выделения цепня. Бычий цепень живет до 18 лет, производя за этот период до 11 млрд яиц.

Вооруженный (свиной) цепень (*Taenia solium*), или свиной солитер, немного уступает бычьему по своим размерам — 2—4 м. Дефинитивным хозяином является человек. На головке помимо четырех присосок имеется хоботок с венчиком острых хитиновых крючьев. Матка зрелого членика имеет всего 8—12 ответвлений с каждой стороны, что определяет меньшее число зрелых яиц в ней — не более 50 тыс. Цикл развития > I ого цепня сходен с циклом развития бычьего цепня. Однако свиной солитер для человека более опасен, так как его труднее изгнать из кишечника (он прочно прикреплен к стенке кишки), а главное, человек может быть и промежуточным хозяином. Финны солитера развиваются

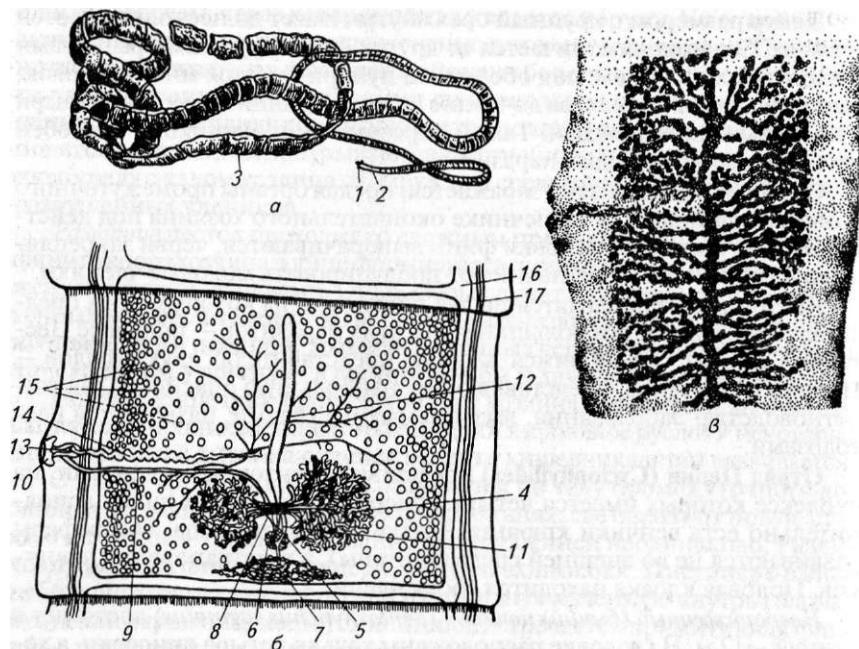


Рис. 56. Цепень невооруженный:
 а — внешний вид; б — гермафродитный членник; в — зрелый членник; 1 — головка;
 2 — шейка; 3 — стробила; 4 — двураздельный яичник; 5 — яйцевод; 6 — тельце Мелиса;
 7 — желточник; 8 — семяприемник; 9 — влагалище; 10 — половая клоака; 11 — устье матки;
 12 — матка; 13 — копулятивный орган; 14 — семяпровод; 15 — семенники; 16 — канал выделительной системы; 17 — нервный тяж

ся в различных внутренних органах, в том числе в печени, сердце, мозге, что может привести к смерти.

Зрелые членники из тела человека могут выходить целыми обрывками стробилы. Членники передвигаться по субстрату не могут. Промежуточным хозяином может быть свинья, кабан, собаки, кошки, кролики, зайцы, медведь, верблюд, иногда и человек. Финны типа цистицерк концентрируются в основном в мышцах, но могут оседать в сердце, печени, мозге и глазах, длительное время (до 6 лет) сохраняя жизнеспособность. Заражение происходит при поедании непрожаренного и непроваренного мяса, чаще всего свиного. Человек может быть промежуточным хозяином, если в его кишечник попадут зрелые яйца. В этом случае финны образуются в мышцах человека, вызывая тяжелое заболевание. Взрослые черви живут в кишечнике человека несколько лет.

Овечий мозговик (Multiceps multiceps) — червь небольших размеров; длина его доходит до 80 см. Головка кроме четырех присосок вооружена хоботком с двумя рядами крючьев. Дефинитивным хозяином является собака и ее дикие родичи. Зрелые членники с калом собак попадают

мо внешнюю среду. Если яйца будут проглочены промежуточным хо-
| ином (овца, коза, а также крупный рогатый скот, реже свиньи, верб-
люды и другие животные, очень редко человек), то из яиц выходят он-
косфера, которые внедряются в стенки кишечника и с током крови
разносятся по организму овцы. В головном мозге животного онкосфе-
ра превращается в центр, достигающий размеров куриного яйца. По-
раженная финной овца совершает круговые движения, так как обычно
Юражается одна половина мозга, что и определило название болезни —
нертнячка овец. Среди больных овец наблюдается массовая гибель. Соба-
ки и заражаются, поедая мозг погибших от вертнячки овец. Взрослые черви
живут до 6—8 мес.

Мониезии (различные виды рода *Moniezia*) достигают 5 м в длину и
более. Головка червя имеет только четыре присоски. Особенностью
мониезии является то, что в каждом гермафродитном членике находит-
ся двойной комплект половых органов, а половые отверстия расположены
по обеим сторонам членника. Дефинитивным хозяином паразита
является мелкий и крупный рогатый скот. Особенно тяжело переносят
заболевание молодые животные.

Промежуточными хозяевами служат некровососущие микроскопиче-
ские малые панцирные клещи, населяющие почву. Клещи поедают
онкосферы, выпавшие из разрушившихся членников паразита. Онко-
сфера через стенки кишечника клещей проникают в полость тела и там
превращаются в мелкие финны типа цистицеркоид (мельчайшая личинка
с одной головкой). Млекопитающие заражаются, поедая с травой
пораженных клещей. В борьбе с мониезией важен режим чередо-
вания выпаса скота на пастбищах.

Карликовый цепень (*Nyumentolepis nana*) соответствует своему названию:
его длина около 1 см. Развитие червя происходит в одном хозяине — че-
ловеке. В кишечнике человека может паразитировать до тысячи парази-
тов одновременно. Из онкосфер, попавших в кишечник человека, в
ворсинках развиваются мелкие финны — цистицеркоиды, которые за-
тем выпадают в просвет кишечника, закрепляются с помощью присо-
сок и крючьев на стенках кишки и превращаются в половозрелых чер-
вей. У человека часто наблюдается самозаражение. Весь цикл развития
цепня занимает всего 20 сут.

Эхинококк (*Echinococcus granulosus*) достигает в длину около 5 мм
(рис. 57). Головка этого паразита имеет четыре присоски и хоботок с
двумя рядами крючьев. Для эхинококка характерно наличие всего
трех-четырех членников: незрелый, гермафродитный и зрелый. Зрелый
членник, содержащий до 800 яиц, отрывается от тела паразита и выносится
с каловыми массами дефинитивного хозяина (собаки, волка, шакала,
лисицы, а также других хищных животных) во внешнюю среду. Зрелые
членники во внешней среде некоторое время могут передвигаться, в том
числе и в шерсти хозяина. Место оторвавшегося зрелого членника после
оплодотворения занимает гермафродитный членник, становясь зрелым.

Промежуточным хозяином могут стать мелкий и крупный рогатый
скот, свиньи, лошади, кролики, грызуны и другие млекопитающие,

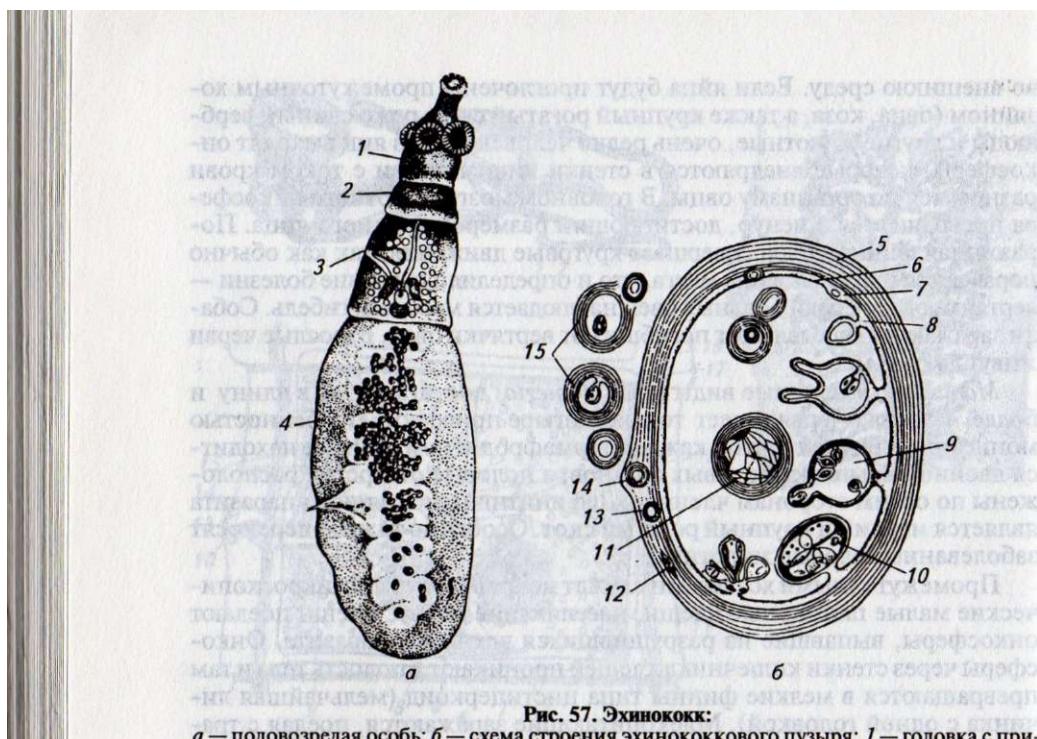


Рис. 57. Эхинококк:
а — половозрелая особь; б — схема строения эхинококкового пузыря; 1 — головка с присосками и хоботком, вооруженным крючьями; 2 — незрелый членик; 3 — гермафродитный членик; 4 — зрелый членик; 5 — кутикула; 6 — производящая оболочка; 7, 8, 9, 10, // — внутренние дочерние пузыри на разных стадиях развития; 12, 13, 14, 15 — наружные дочерние пузыри на разных стадиях развития

кроме семейства собачьих. Может им стать и человек. Заболевание называется эхинококкозом, часты летальные исходы.

В кишечнике промежуточного хозяина из яйца выходит онкосфера. Попадая через стенки кишечника в кровяное русло, онкосфера мигрирует по всему организму, оседая и образуя финны типа эхинококк чаще всего в печени, реже в легких, мышцах и мозге. У крупного рогатого скота масса финны может достигать более 50 кг. Внутри финны много дочерних пузырей, что можно рассматривать как бесполое размножение паразита на ранних стадиях развития.

Источником заражения человека могут стать собаки, особенно пастушки или при свободном содержании в поселках и городах. Выползающие из анального отверстия зрелые членики вызывают у собак зуд, животные чешутся и разносят яйца по шерсти. Взрослые черви живут в кишечнике собак до 6 мес. Эхинококк распространен в местностях с развитым животноводством.

В последние годы участились случаи заражения человека альвеококком (*Alveococcus multilocularis*) в Европейских странах. Дефинитивным хозяином этого мелкого цепня, похожего на эхинококка, обычно являются

лисицы, собаки и кошки. Промежуточным хозяином могут быть мышевидные грызуны, которые заражаются яйцами альвеококка, подбирая остатки у нор лис и жилищ человека, где могут встречаться зараженные кошки и собаки. Человек также может стать промежуточным хозяином, заражаясь яйцами цепня от собак и кошек. Чаще всего финны образуются у человека в дыхательных путях, что может вызывать удушье.

Отряд лентецы (Pseudophyllidea). У представителей этого отряда головка не имеет присосок и крючьев. Органами прикрепления служат ботрии — щелевидные ямки, с помощью которых паразиты защемляют стенку кишечника. Матка у лентецов открывается наружу на брюшной стороне членика отверстием, через которое зрелые яйца могут выходить в просвет кишечника дефинитивного хозяина. Отряд включает много паразитических видов, из которых наиболее опасен широкий лентец (*Diphyllobothrium latum*). Червь живет в кишечнике хищных животных, достигая длины 8—10 м. Дефинитивным хозяином может быть человек, заражаются также дельфины и тюлени. Вышедшие с калом во внешнюю среду зрелые яйца с крышечкой должны попасть в пресную воду. Из яиц в воде выходит личинка, покрытая ресничками, — корацидий (рис. 58). Личинку могут съесть веслоногие ракообразные —

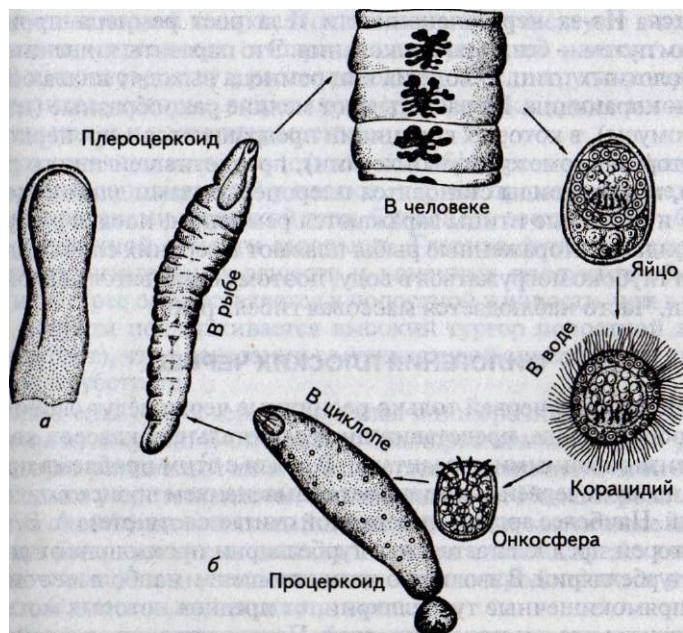


Рис. 58. Лентец широкий *Diphyllobothrium latum*:
а — головка; б — цикл развития

цикlopы и диаптомусы, в кишечнике которых из корацидия выходит сформировавшаяся личинка онкосфера с шестью крючьями. Онкосфера внедряется в полость тела первого промежуточного хозяина и там превращается в покоящуюся фазу — процеркоид, имеющий удлиненную форму с диском на заднем конце тела, несущем крючья.

Если пораженного ракообразного проглотит рыба (щука и другие), то в теле рыбы процеркоид внедряется во внутренние органы и мышцы, где превращается в финнозную стадию — плероцеркоид. Плероцеркоиды имеют червеобразное тело длиной 1—2 см с одной ввернутой головкой на переднем конце тела. Головка вооружена двумя ботриями.

Дефинитивный хозяин заражается, поедая сырую, замороженную или слабо просоленную рыбу. Человек заболевает дифиллоботриозом чаще всего там, где потребляют в больших количествах сырую рыбу, в основном в виде строганины из замороженной рыбы, в которой остаются жизнеспособные плероцеркоиды. В цикле развития лентеца важны резервуарные хозяева (хищные рыбы). При поедании щуками пораженных плероцеркоидами рыб в кишечнике щук паразиты не перевариваются, а проникают в их ткани и накапливаются в различных органах.

К лентециам относится червь, имеющий нерасчлененное тело, — *ремнец* (сем. Ligulidae). Длина паразита колеблется от 7 до 200 см. Тело в виде ленты, в которой комплекты половых органов многократно повторяются. Из-за нерасчлененности тела рост ремнца происходит обычным путем — без зоны почкования. Это паразиты кишечника водных и болотных птиц. В воде из яиц ремнца выходят плавающие личинки — корацидии. Их заглатывают мелкие ракообразные (цикlopы и диаптомусы), в которых корацидии превращаются в процеркоиды. В рыбе (второй промежуточный хозяин), проглатившей такого ракообразного, процеркоиды становятся плероцеркоидами длиной до 60 см. Водные и болотные птицы заражаются ремнцем, поедая рыбу с плероцеркоидами. Пораженные рыбы плавают в верхних слоях воды; они не могут глубоко погружаться в воду, поэтому становятся легкой добычей птиц. Часто наблюдается массовая гибель рыб.

ФИЛОГЕНИЯ ПЛОСКИХ ЧЕРВЕЙ

Среди плоских червей только ресничные черви ведут свободноживущий образ жизни, представители всех остальных классов являются специализированными паразитами. В связи с этим проблема происхождения плоских червей ограничивается выяснением происхождения турбеллярий. Наиболее аргументированной считается гипотеза А. В. Иванова, в которой предполагается, что турбеллярии произошли от ацелоподобных турбеллярий. В эволюционном отношении наибольшее значение имеют прямокишечные турбеллярии, от предков которых могли произойти другие классы плоских червей. Переход плоских червей к паразитизму мог осуществляться через симбиоз, тем более что такие тенденции проявляются и у современных турбеллярий.

Моногеней могли произойти от турбелляриеподобных предков через квартирантство на жабрах и плавниках рыб. Затем постепенно они перешли к эктопаразитизму. Среди современных моногеней наблюдается переход к эндопаразитизму, например у лягушачьей многоустки. Родственные связи моногеней и цестод были доказаны русским ученым И. Е. Быховским. Эволюция же трематод могла идти независимо от моногеней и цестод. Можно предположить, что на первом этапе эволюции трематоды вели свободный образ жизни, а к паразитизму перешли их личинки, вступившие в симбиотические отношения с моллюсками.

ТИП КРУГЛЫЕ, ИЛИ ПЕРВИЧНО- ПОЛОСТНЫЕ ЧЕРВИ (*Nemathelminthes*)

Круглых червей часто называют первично-полостными червями, так как они имеют несегментированное тело с первичной полостью, заполненной полостной жидкостью. Кишечный канал не разветвлен и заканчивается анальным отверстием. Известно более 100 тыс. видов круглых червей, среди которых много свободноживущих форм, встречающихся в морях, пресных водоемах и в почве. Практическое значение круглых червей велико. Почвенные виды участвуют в почвообразовательных процессах, а паразитические черви наносят огромный ущерб животноводству и растениеводству. Среди круглых червей много паразитов, которые встречаются практически у всех многоклеточных животных и у многих растений.

Первичная полость тела (схицоцель) образуется за счет разрушения паренхимы, заполняющей у плоских червей промежутки между внутренними органами и стенкой тела. Схицоцель соответствует бластоцелю — первичной полости зародыша. Главная функция схицоцеля — транспорт питательных веществ и конечных продуктов обмена, что легче и быстрее осуществляется в полостной жидкости, чем в паренхиме. В полости поддерживается высокий тургор полостной жидкости (гидроскелет), что в сочетании с мускулатурой способствует движению нематод в субстрате.

Форма тела у этих червей округлая в поперечнике. Покровы представлены кутикулой. В отличие от брюхоресничных червей у нематод отсутствует ресничный эпителий. Остатки ресничного эпителия имеются только у представителей низших классов. Под покровами располагается слой продольных мышц или отдельные мышечные пучки, которые могут быть кольцевые мышцы. Нервная система представлена окончательным узлом и отходящими от него продольными нервыми тяжами. Органы дыхания отсутствуют.

Кишечник состоит из трех отделов: переднего, среднего и заднего. Переднее отверстие находится на брюшной поверхности переднего конца тела. Имеется анальное отверстие.

Выделительная система построена по протонефридиальному типу, но без мерцательных клеток, или в виде особых кожных (гиподермальных) желез.

Большая часть круглых червей раздельнополые, но встречаются и гермафродитные формы. Часто выражен половой диморфизм. Размножаются толькоовым путем. Развитие прямое, реже — с метаморфозом. Круглые черви не способны к регенерации.

Тип Круглые черви включает несколько классов, из которых наибольший интерес представляют три класса: класс Собственно круглые черви, или Нематоды (*Nematoda*), класс Скребни (*Acanthocephala*) и класс Коловратки (*Rotatoria*). Для сельского хозяйства наибольшее значение имеют представители класса нематод.

КЛАСС СОБСТВЕННО КРУГЛЫЕ ЧЕРВИ, ИЛИ НЕМАТОДЫ (*Nematoda*)

Общая характеристика. Округлое тело длиной от нескольких миллиметров до 1 м (паразитическая нематода кашалота достигает в длину 8 м) сохраняет постоянную ширину и покрыто плотной кутикулой. Свободноживущие виды населяют соленые и пресные воды, живут в почве. Много паразитов растений, животных и человека. Только паразитов растений описано более 1 тыс. видов. Среди паразитических форм большой интерес представляют те виды, которые паразитируют на вредных насекомых и сорных растениях.

Строение и жизненные отправления. Несмотря на большое экологическое разнообразие, нематоды однообразны морфологически. Форма тела обычно веретенообразная, нитевидная и реже колбасовидная. Свободноживущие черви живут в илистом грунте водоемов или гумусовом горизонте почвы, питаясь органикой. Паразитические формы, обитаю в тканях растений и в теле животных, поглощают органические вещества из организма хозяина. Есть виды с округлой формой тела. На переднем конце расположены рот, органы осязания и химического чувства, светочувствительные органы встречаются редко. Туловище заканчивается хвостовым отделом, следующим за анальным отверстием. Тело у самцов может оканчиваться бурсой — органом прикрепления к телу самки при спаривании. Половой диморфизм часто хорошо выражен.

Покровы нематод образованы гиподермой, покрытой кутикулой. Кутикула может состоять из четырех—десяти слоев. Ее поверхность кольчатая или гладкая. Химический состав кутикулы представлен сложном комплексом белков, липопротеинов и других веществ. Кутикула благодаря своему составу находится в биологически активном состоянии и устойчива к действию пищеварительных ферментов хозяина, хотя у погибших червей кутикула легко переваривается в кишечнике животных.

Вещества, образующие кутикулу, выделяются клетками гиподермы. У свободноживущих червей гиподерма представлена однослойным эпителием, а у паразитических взрослых червей — протоплазматической массой, содержащей многочисленные ядра. На спинной и брюш-

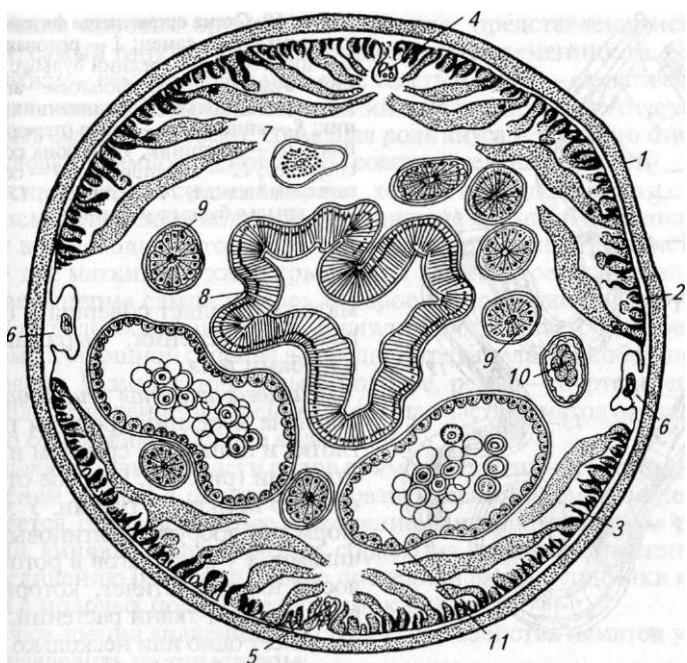


Рис. 59. Поперечный разрез тела самки аскариды *Ascaris lumbricoides*:
1 — кутикула; 2 — гиподерма; 3 — мышцы; 4 — спинной валик гиподермы с нервным ганглием; 5 — брюшной нервный тяж; 6 — боковые валики гиподермы с каналами выделительной системы; 7 — первичная полость тела; 8 — кишечник; 9 — яичники; 10 — яйцевод; 11 — матка

ной сторонах, а также по бокам гиподерма образует валикообразные утолщения, тянувшиеся вдоль тела.

Первичная полость тела заполнена полостной жидкостью и заключена в кожно-мышечный мешок. В полости тела расположены внутренние органы,

Нервная система в виде окологлоточного нервного кольца, которое опоясывает пищевод, и отходящих от него двух продольных нервных тяжей: спинного и брюшного. Образована нервная система небольшим числом нервных клеток. Большее развитие получили брюшные и спинные нервные тяжи, соединенные комиссурами и дающие ответвления к различным органам (рис. 59). Органы чувств развиты слабо и представлены осязательными и обонятельными клетками.

Мускулатура образована обычно полосами продольных мышечных волокон, которые разделены по бокам тела выростами (валиками) гиподермы, а на спинной и брюшной сторонах — нервыми стволами, нежащими в валиках гиподермы. Мышечные полосы образованы слоем удлиненных клеток, содержащих миофibrиллы. Мускулатура не-

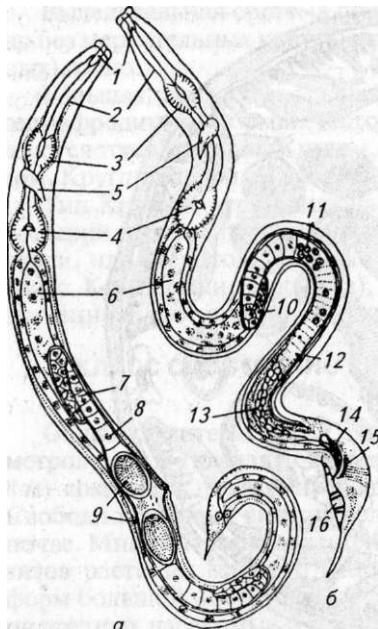


Рис. 60. Схема организации фитонематод:
а — самка; б — самец; 1 — ротовая полость;
2 — пищевод; 3 — средний бульбус; 4 — зад-
ний бульбус с дробильным аппаратом;
5 — нервное кольцо; 6 — кишечник; 7 — яич-
ник; 8 — яйцевод; 9 — матки (передняя и зад-
няя); 10 — семенник; 11 — зона созревания
спермиев; 12 — семяпровод; 13 — семязвер-
гательный канал; 14 — спикулы; 15 — рулек;
16 — крылья бурсы с ребрами

матод позволяет совершать однообразные движения, выражющиеся изгибами тела.

Пищеварительная система представлена передним (ротовая полость, глотка и пищевод), средним и задним отделами (рис. 60). Ротовое отверстие обычно прикрыто губами. У хищных форм рот вооружен хитиновыми зубчиками, а у фитофагов в ротовой полости имеется стилет, которым они прокалывают ткани растений. Пищевод имеет одно или несколько расширений (бульбусов) с мощной мускулатурой. Передний отдел кишечника выстлан эпителием.

Процессы переваривания пищи происходят в средней части кишки, выстланной однослойным эпителием эндоцермального происхождения. Задняя кишка выстлана кутикулой и заканчивается анальным отверстием.

Органы дыхания. У свободноживущих нематод и фитопаразитов газообмен происходит через покровы тела; у большинства паразитов животных и человека анаэробное дыхание.

Органы выделения. В головном отделе нематод расположены одна-две гигантские клетки гиподермы, называемые шейными железами, которые выполняют функции осморегуляции и выделения. Железа имеет отростки и канал внутри, который открывается наружу. У многих нематод шейная железа представляет собой большую клетку с двумя отростками, которые тянутся вдоль тела в боковых валиках гиподермы и имеют внутри каналы. Эти каналы в шейной железе сливаются и открываются наружу порой на брюшной стороне в передней части нематоды. Функцию почек накопления выполняют одна-две пары фагоцитарных клеток, лежащих около выделительных каналов.

Органы размножения. Нематоды раздельнополы. У них обычно развит половой диморфизм. У паразитических нематод самки крупнее самцов, задний конец тела которых закручен. У части фитонематод самки при созревании яиц сильно раздуваются, приобретая округлую форму. Половые органы имеют трубчатое строение.

Мужские половые органы, как правило, представлены непарной трубкой, самый тонкий конец которой является семенником. Средняя часть трубы — семяпровод, а наиболее толстый отдел — семязвергательный канал, который открывается в задний отдел кишечника. Анус у самцов выполняет функцию клоаки, совмещающую роль ануса и полового отверстия. (около клоаки у самцов расположены совокупительные спикулы.

Женская половая система парная, хотя есть представители с непарной женской половой системой. Нитевидные яичники постепенно переходят в яйцеводы, которые расширяются и переходят в толстые каналы — две матки. Матки открываются в непарное влагалище, куда поступает сперма самца и через которое в последующем наружу выводятся оплодотворенные яйца. Влагалище открывается половым отверстием на брюшной стороне в передней трети тела на особом перехвате — пояске. Размножение только половое, редко — партеногенетическое. Оплодотворение внутреннее. Большая часть нематод откладывают яйца, но есть и живородящие виды.

Развитие у большей части видов прямое, у других — с метаморфозом и со сменой хозяев. У некоторых паразитических форм в цикле развития имеется гермафродитное поколение. Личинки в период роста и развития линяют, периодически сбрасывая кутикулу, препятствующую увеличению размеров. После последней линьки личинки превращаются в молодых половозрелых самок и самцов.

С точки зрения значения для сельского хозяйства нематод условно можно разделить на три группы:

- свободноживущие в водоемах и почвах сапрофитные и хищные формы, питающиеся органическими остатками и представителями мелкой почвенной фауны;
- паразиты растений, обитающие в тканях различных растений в течение всей жизни или на определенной стадии своего развития;
- ® паразиты животных и человека.

Паразитических нематод условно можно разделить на две экологические группы: геогельминты, часть жизненного цикла которых проходит во внешней среде, и биогельминты, развитие которых проходит только в одном или нескольких хозяевах без выхода во внешнюю среду.

Свободноживущие круглые черви. В соленых и пресных водах, а также в почве обитает множество свободноживущих мелких круглых червей. Водные нематоды являются важным звеном в цепях питания юных животных. Для сельского хозяйства большой интерес представляют нематоды, населяющие почву. Они предпочитают хорошо увлажненные и богатые органикой почвы. В благоприятных условиях обитания в 1 м² верхнего почвенного слоя можно насчитать десятки миллионов этих червей. Основная их часть — постоянные обитатели почв, но значительное их число находится в почве только на определенной стадии своего развития (жизненного цикла).

Почвенные нематоды питаются гниющими остатками растений и других органических веществ. Есть и хищные нематоды, поедающие других мелких почвенных животных. Все почвенные нематоды явля-

ются участниками почвообразовательного процесса. Перерабатывая органику, нематоды способствуют гумификации почв. Зная биологические особенности почвенных нематод и изучив закономерности их жизненных циклов, полеводы могут создавать благоприятные условия для их жизнедеятельности, используя соответствующую технику, систему обработки почв, их мелиорации, внесения органических и минеральных удобрений и т. п. Зная особенности жизнедеятельности хищных нематод, можно попытаться использовать их для борьбы с вредными животными, в частности с фитонематодами.

Круглые черви — паразиты растений (фитонематоды). Для круглых червей, паразитирующих на растениях (фитонематоды), характерны небольшие размеры тела (длина 0,1—12 мм, ширина около 15—20 мкм), имеющего цилиндрическую, нитевидную или веретенообразную форму. Часто самки уплощены и по форме напоминают мешок или лимон. Тело состоит из трех отделов: головного, собственно тела и хвостового. В ротовой полости нематод имеется колющий орган — стилет (копье), с его помощью нематода прокалывает ткани растений. В стилете находится канал, через который нематода впрыскивает секрет желез пищевода в растение. Общим для большинства фитонематод является частично внекишечное пищеварение. В секрете желез пищевода содержатся пищеварительные ферменты, которые вызывают гидролиз высокомолекулярных соединений. Приготовленная полупереваренная питательная смесь засасывается с помощью стилета в кишечник фитонематоды, где заканчивается процесс переваривания и всасывания пищи. В средней части пищевода имеется расширение — бульбус, с его помощью паразит всасывает соки растений (рис. 61).

При движении фитонематоды совершают медленные волнообразные движения. Есть и неподвижные формы; это — самки, тип питания которых не требует перемещения. У таких нематод мускулатура полностью редуцирована, подвижна лишь передняя часть тела. Нематоды могут передвигаться на небольшие расстояния — в пределах 30 см, но



0
δ

Рис. 61. Формы стилета и копья фитонематод:
 $a - 2$ — стилеты; $\delta - 3$ — копья

существуют формы и более подвижные, передвигающиеся на расстояния до 100 см.

Фитонематоды раздельнополы, для них характерен половой диморфизм. Особенно сильно половой диморфизм выражен у видов, имеющих раздутых сидячих самок, характеризующихся мешкообразной формой тела. Самцы же имеют типичную для нематод нитевидную форму и способны передвигаться.

Размножаются фитонематоды половым путем, очень редко встречается гермафроптизм и партеногенез. Размножение всегда происходит яйцами, есть виды, у которых наблюдается живорождение — личинки выходят из яиц еще в яичнике самки. Оплодотворение внутреннее. У самца имеются специальные выросты дна клоаки — спикулы. Их назначение — расширение вульвы при совокуплении. Спермин фитонематод совершают амебоидное движение, так как они лишены хвоста.

Плодовитость фитонематод высока; одна самка может отложить несколько сотен яиц, но у отдельных видов это число доходит до нескольких тысяч. Следует различать два показателя плодовитости: число яиц, отложенных самкой за всю ее жизнь, и число яиц, развивающихся в матке самки одновременно.

Нематоды откладывают яйца в соответствии со своим образом жизни: в ткань растений, в яйцевой мешок на субстрате, внутрь галла (вздутия в месте повреждения паразитом), яйца могут оставаться в теле самки; в этом последнем случае тело ее представляет собой нечто вроде цисты.

Механизм выхода личинки из яйца весьма сложен и до конца не ясен. Есть виды, у которых выход личинок стимулируется выделениями корней растений-хозяев. У других представителей выход личинок зависит от влажности, температуры, содержания кислорода и т. п. Перед выходом личинка в яйце совершает быстрые движения, разрывающие оболочку яйца. Развитие личинок включает четыре личиночных возраста и взрослую форму. Все эти стадии четко разграничены линьками.

Личинки первого возраста у ряда видов проходят линьку еще в яйце. Такой тип первой линьки называют закрытым (денударным), и он характерен для фитонематод. Сапробиотические нематоды первую линьку проходят уже вне яйца — открытый тип (конвелярный).

Личинки первого возраста отличаются от взрослых форм (имаго) в основном значительно меньшими размерами, а также недоразвитыми половой и пищеварительной системами. Зачаточная половая система появляется лишь у личинок второго возраста, и затем по мере роста и развития личинок она достигает своего окончательного развития.

Перед началом линьки личинки перестают питаться. В процессе линьки вся кутикула сбрасывается и одновременно формируется новая. В период развития у нематод может меняться соотношение полов: при неблагоприятных условиях внешней среды (нетипичное растение-хозяин, засуха, нематодоустойчивые сорта) развивается больше самцов. Число генераций у нематод, имеющих несколько поколений, также зависит от условий среды, и прежде всего от температуры и влажности.

Онтогенез фитонематод обусловлен особенностями их связи с растением-хозяином. Одна группа фитонематод-эктопаразитов питается, проникая стилетом в растение, но весь жизненный цикл их проходит в почве, где они мигрируют (лонгидоры, триходоры и др.).

У группы нематод—полуэндопаразитов корней (спиральные нематоды) часть жизненного цикла проходит в почве, где они мигрируют, а часть — в корнях растений, куда они внедряются передним концом тела.

Эндопаразитические формы фитонематод развиваются в корнях растений, но могут свободно мигрировать из корней в почву и обратно, т. е. почва для этих нематод является средой переживания (пратиленхи).

Онтогенез нематод—эндопаразитов корней (тиленхиды, гетеродериды и др.) проходит только в тканях растений без выхода паразитов в почву. Растения для этих фитонематод являются постоянной средой обитания. Нематод, поражающих надземные части растений, по особенностям онтогенеза также можно условно разделить на три группы. Есть эктоэндопаразиты, которые мигрируют на растении, почва для этих нематод не нужна. У второй группы эндопаразитических нематод (стеблевые нематоды), которые мигрируют только в тканях растений, а также группы галлообразователей (ангвины), онтогенез проходит в одном локальном месте растения. Почва для этих двух последних групп служит средой переживания.

Систематика вредоносных фитонематод базируется на различиях в их морфологии, биологии и на особенностях их экологии:

Класс Круглые черви (Nematoda)

Подкласс Сецерненты, или Фазмидиевые (Secernentea)

Отряд Тиленхиды, или Настоящие шишкоиглые нематоды (Tylenchida)

Отряд Афеленхиды (Aphelenchida)

Подкласс Аденофореи (Adenophorea)

Отряд Дорилаймиды (Dorylaimida)

Небольшое число фитонематод относятся к отряду Афеленхиды. Эти нематоды повреждают надземные части растений (завязи, почки, листья) и могут быть как экто-, так и эндопаразитами. *Рисовый афеленхойд* (*Aphelenchoides besseyi*) является полифагом, но наибольший вред приносит рисовым и земляничным посадкам. Нематоды разного возраста сохраняются под пленкой рисовой зерновки и в послеуборочных остатках. Весной нематоды выходят в почву и отыскивают всходы риса. *Земляничная нематода* в симбиозе с бактерией *Corynebacterium fascians* вызывает заболевание земляники «цветная капуста».

Хризантемная нематода (*Aphelenchoides ritzemabosi*) является эктопаразитом хризантем, земляники, крыжовника, смородины, томата, ма-лины и других растений. Паразиты зимуют на стеблях или в почве.

В отряде Дорилаймиды подкласса Аденофореи имеется несколько видов нематод, являющихся паразитами растений. Крупные нематоды из семейства Лонгидориды повреждают многие культуры: овощные, свеклу, табак, зерновые, бобовые травы, землянику, древесные и т. д.

Основная масса червей, паразитирующих в растениях (фитонематод) относится к отряду 111ишкоиглые нематоды подкласса Сецирненты. В этот подкласс входят также нематоды почвенные, сапробиотические, хищные, паразиты насекомых.

Нематод из семейства Гетеродериды часто называют разнокожими нематодами, так как самки гетеродерид имеют шаровидное тело с плотными, часто окрашенными покровами, а самцы — вытянутое гонкое тело с прозрачными покровами. У нематод сильно развит стилет. У самок парные яичники, у самцов один или два семенника.

Подсемейство Гетеродерины относится к группе цистообразующих нематод, у которых в жизненном цикле обязательно наличие особых образований из отмершего тела самки — цист. В цистах яйца и инвазионные личинки в течение нескольких лет сохраняют жизнеспособность даже при неблагоприятных условиях.

Биологический цикл у всех видов цистообразующих нематод одинаков (рис. 62). Из перезимовавших в почве цист весной выходят личинки второго возраста. Они поражают корни растений, которые часто находят посредством хемотаксиса на выделения корней растения-хозяина.

В корнях личинки становятся неподвижными, усиленно питаются, линяют и превращаются в бутылковидных личинок третьего возраста. После следующей линьки они превращаются в сильно утолщенных личинок четвертого возраста, принимающих шаро-, лимоно-, груше- или мешкообразную форму. У них подвижен только головной отдел. Из личинок четвертого возраста развиваются самки и самцы в соотношении 1:1. При неблагоприятных условиях развития доля самцов возрастает.

Самцы в оболочке личинки находятся в свернутом состоянии. После разрыва шкурки самцы выходят через разрыв корня растения в почву. Тело самцов тонкое и прозрачное, что не позволяет обнаружить их невооруженным глазом.

Самки также разрывают ткани коры корня растения, и в этом разрыве находится задний конец их тела. Головной же отдел погружается в ткань корня. Таким образом, самки как бы прикреплены к корням, и их крупное (до 1 x 0,5 см) белое тело легко обнаружить невооруженным глазом. Самцы находят самок, оплодотворяют их и после этого погибают, хотя у некоторых видов один самец может оплодотворить нескольких самок. Самки некоторых видов вырабатывают половые аттрактанты, играющие важную роль в привлечении самцов.

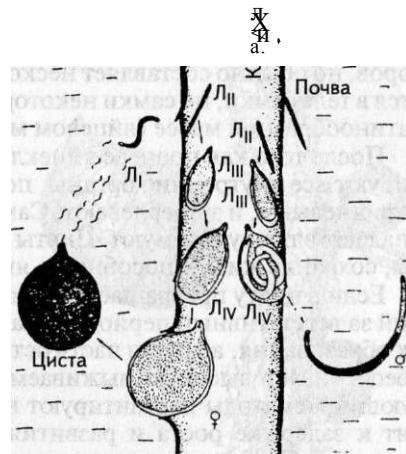


Рис. 62. Цикл развития цистообразующих нематод: L₁—L₄ — личиночные стадии

Оплодотворенные самки питаются и начинают откладывать яйца, число которых зависит от наличия пищи и других абиотических факторов, но обычно составляет несколько сотен. Большая часть яиц остается в теле самки, но самки некоторых видов откладывают наружу в желатинообразной массе (яйцевом мешке) по нескольку яиц.

После того как процесс яйцекладки завершен, в теле самки регенерируют все внутренние органы, покровы тела меняют цвет, становясь коричневыми, и затвердевают. Самки превращаются в цисты, которые опадают в почву и зимуют. Цисты могут находиться в почве долгие годы, сохраняя жизнеспособность яиц и личинок.

Если в цисту превращается самка вида, дающего несколько генераций за вегетативный период, то часть личинок выходит сразу же после их образования, а другая часть остается в цистах на несколько лет, что обеспечивает высокую выживаемость фитонематод. Все цистообразующие нематоды паразитируют внутри корней растений, что приводит к задержке роста и развития растений-хозяев. Степень ущерба культуре обусловлена численностью паразита.

Из цистообразующих фитонематод значительный вред приносят овсяная нематода (*Heterodera avenae*), люцерновая нематода (*H. medicaginis*), соевая нематода (*H. glycines*), свекловичная нематода (*H. schachtii*), картофельная нематода (*H. rostochiensis*) и некоторые другие.

Овсяная нематода (*H. avenae*) является представителем семейства разнокожих нематод (Heteroderidae). Это вредитель злаковых культур, прежде всего овса, а также пшеницы и ячменя. Известны случаи, когда растениями-хозяевами служили пырей, овсяница и дикий овес. Растения, пораженные овсянной нематодой, отстают в росте, их листья рано желтеют. Ущерб особенно велик в засушливые годы, так как гельминты задерживают поступление воды и питательных веществ в растение.

Как все представители цистообразующих нематод, овсяная нематода обладает четким половым диморфизмом. Лимоновидные самки покрыты беловатой слизистой оболочкой. Постепенно кутикула взрослой самки темнеет, и самка превращается в бурую цисту. Самцы имеют нитевидное прозрачное тело длиной 1,2—1,4 мм.

Для цикла развития овсянной нематоды характерно наличие только одной генерации в год. Весной из цист нематод, перезимовавших в почве, начинается выход личинок второго возраста. Они внедряются в корни проростков овса и начинают питаться. После линьки самцы приобретают нитевидную форму, а самки становятся лимоновидными.

Оплодотворенная самка начинает производить яйца. Формирование цист происходит в почве на глубине 10—40 см. Цисты должны перезимовать, так как активация личинок в яйцах цисты становится возможной после прохождения периода низких температур.

Бороться с овсянной нематодой можно путем использования севооборотов с возвратом посевов зерновых злаков не ранее чем через 4 года, а также выведением устойчивых к нематодам сортов зерновых злаков.

Картофельная нематода (*H. rostochiensis*) паразитирует на корнях и клубнях картофеля. Является одним из самых опасных вредителей кар-

гофеля; объект карантинных мероприятий. Из европейской части РФ паразит постепенно расселяется на восток. Поражает растения только из семейства пасленовых — картофель, томат, баклажан. Растения сильно угнетены, листья рано желтеют и завядают, урожай может снижаться на 80 %.

У самца тонкое прозрачное тело длиной не более 1,2 мм. Самки шаровидной формы диаметром до 1 мм. По мере старения покровы самки окрашиваются в коричневый цвет, иногда становясь почти черными. Самки живут на корнях картофеля и выглядят мелкими желтовато-коричневыми шариками (рис. 63), у самок яйцевых мешков не образуется.

В одной цисте может быть до 1200 яиц. Личинки нематод выходят из цист и проникают в корни всходов картофеля. На корнях образуются вздутия, а через месяц растущие личинки разрывают кору корня, линяют последний раз и становятся взрослыми червями. Самцы выходят в почву для поиска самок и спаривания с ними. У оплодотворенных самок тело становится шаровидным, его полость заполняется яйцами.

Для выхода личинок из цист необходимы не только оптимальные температура и влажность, но и наличие растения-хозяина. Выделения корней растения-хозяина стимулируют выход личинок. Новая генерация личинок появляется через 40—75 сут, т. е. в течение лета возможно развитие одной, реже двух генераций фитогельминта.

Для борьбы с картофельной нематодой и профилактики заражения рекомендуется использовать противонематодный севооборот с возвращением восприимчивых культур не ранее чем через 3—4 года. Хорошие результаты дает выращивание нематодоустойчивых сортов картофеля.

Свекловичная нематода (H. schachtii) распространена повсюду, но особый вред приносит в южных регионах, где выращивают сахарную свеклу. Лимоновидное тело самки окрашено в темно-желтый или бурый цвет (рис. 64). Размеры тела самок не превышают 0,7—1 мм в длину и 0,4—0,5 мм ширину. Самец с нитевидным прозрачным телом диаметром 0,02—0,03 мм достигает длины 1,5 мм. Стилет мощный, семенник один.

Свекловичная нематода паразитирует на растениях семейств маревых и крестоцветных, но особый ущерб причиняет посевам сахарной

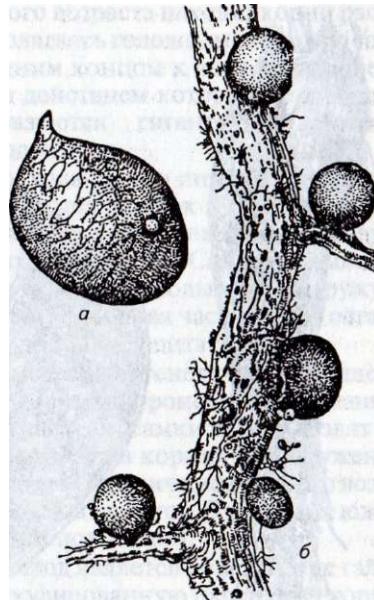


Рис. 63. Самки картофельной нематоды *Heterodera rostochiensis*:
а — циста; б — самки на корнях картофеля

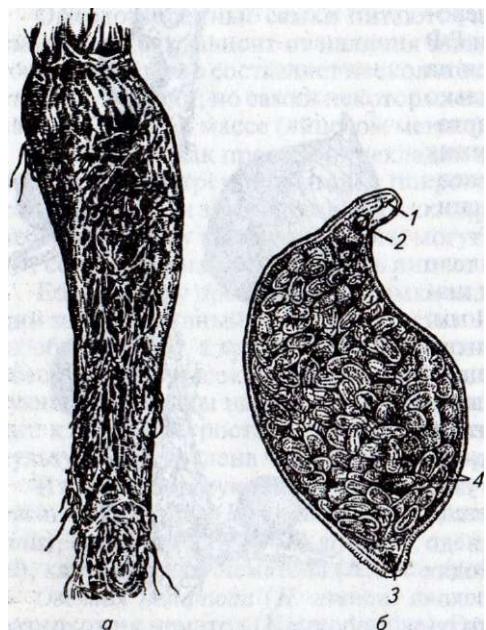


Рис. 64. Свекловичная нематода *Heterodera schachtii*:
а — корневая система свеклы, поврежденная нематодой; б — самка нематоды; 1 — стилет; 2 — средний бульбус; 3 — вульва; 4 — яйца

свеклы. Потери урожая могут достигать 60 % при одновременном снижении (до 15 %) сахаристости. Из-за недостаточного питания пораженные нематодой растения угнетены, их листья желтеют и засыхают. Посевы поражаются пятнами. На участках с большим количеством цист в почве может наблюдаться полная гибель растений.

Самки свекловичной нематоды в отличие от овсяной и картофельной могут откладывать часть яиц в яйцевые мешки и при благоприятных

условиях давать несколько генераций в течение вегетации свеклы. Для развития одного поколения требуется около 4—5 нед. Развитие одной генерации свекловичной нематоды при 18 °C занимает 57 сут, а при 28 °C — 24 сут. По этой причине в Московской области две генерации, а в Винницкой — три-четыре. За свою жизнь самка формирует 100—150 яиц, но иногда и больше. Личинки, которые находятся в яйцах, заключенных в цисты, могут сохранять жизнеспособность до 9 лет.

Выход личинок из цист начинается при температуре 10 °C и выше, но оптимумом считается 18—28 °C. Радиус расселения личинок из цисты не превышает 40 см. Для выхода личинок необходимы не только соответствующие температура и влажность почвы, но и наличие растения-хозяина, секрет корней которого активирует нематод.

Для профилактики рекомендуют бороться с сорными растениями из семейств Маревые и Крестоцветные, соблюдать севооборот с возвращением повреждаемых культур не ранее чем через 5 лет, использовать нематодоустойчивые сорта свеклы, применять в севооборотах растения—антагонисты фитонематоды (вика, клевер и другие бобовые, цикорий, злаковые и др.).

Представители рода Мелойдогини из семейства Мелойдогиниды (*Meloidogynidae*) относятся к галловым нематодам. Особенность галловых нематод в том, что самки у них не превращаются в цисту. Биологический цикл у всех галловых нематод одинаков.

Находящиеся в почве личинки второго возраста находят корни растений-хозяев и внедряются в них, располагаясь головным концом к сосудам проводящих пучков корня, а задним концом к коре корня. Нематода выделяет особые вещества, под действием которых вокруг головной части фитогельминта образуются гигантские клетки. Содержимым этих клеток и питаются паразиты.

Личинки развиваются и линяют, проходя стадии личинок третьего и четвертого возрастов, а затем превращаются в самок и самцов. У нематод средняя часть тела сильно утолщается. Самцы выходят в почву, а самки после последней линьки сильно раздуваются. Самка разрывает кору корня растения и задняя ее часть с вульвой высывается наружу, становясь доступной для оплодотворения. Головная часть самки остается погруженной в корень, где самка продолжает питаться.

Размножаются галловые нематоды и партеногенетически. Общее число яиц достигает сотни и более. Откладка яиц происходит в течение всего вегетационного периода растения. Из тела самки яйца выходят в виде яйцевого мешка. Мешки хорошо видны на корнях невооруженным глазом. Развитие яиц происходит в мешке до личинок второго возраста, которые уходят в почву. Далее цикл повторяется. В теплицах южная галловая нематода может дать до семи поколений.

Важной особенностью галловых нематод является образование галлов. Галлы представляют собой гипертрофированную паренхиму коры корня, которая образуется вокруг внедрившейся личинки в виде вздутия. Размеры галлов могут варьировать от 1 мм до огромных разрастаний. Некоторые растения галлов не образуют. Другая особенность галловых нематод — их широкая пищевая специализация, они полифаги. Известно более 70 видов галловых нематод. В нашей стране практическое значение имеют южная галловая нематода (*Meloidogyne incognita*) и северная галловая нематода (*M. hapla*). Только северная галловая нематода в открытом грунте распространяется далеко на север. Остальные виды встречаются в южных регионах и в тепличных хозяйствах (рис. 65).

Среди представителей семейства Тиленхиды (Tylenchidae), относящихся к настоящим шишкоиглым нематодам, есть широко распространенные вредители сельскохозяйственных культур. Самки, самцы и личинки тиленхид имеют червеобразное тело с заостренными головным и хвостовым концами. Стилет развит слабо. У самок в большинстве случаев один яичник.

Пшеничная нематода (Anguina tritici) поражает все сорта пшеницы. Распространена в южных регионах страны. Из фитогельминтов пшеничная нематода является самым крупным видом: длина половозрелых самок достигает 5 мм при толщине 0,1—0,2 мм. Самцы мельче самок.

Пшеничная нематода живет в тканях надземных частей пшеницы. Типичным признаком повреждения является образование галлов вместо нормального зерна (рис. 66). Зрелые галлы напоминают по форме и размерам зерна пшеницы, но отличаются от них коричневой окраской и шероховатостью. Внутри галла находится белая масса, состоящая в основном из личинок нематод, находящихся в анабиозе.

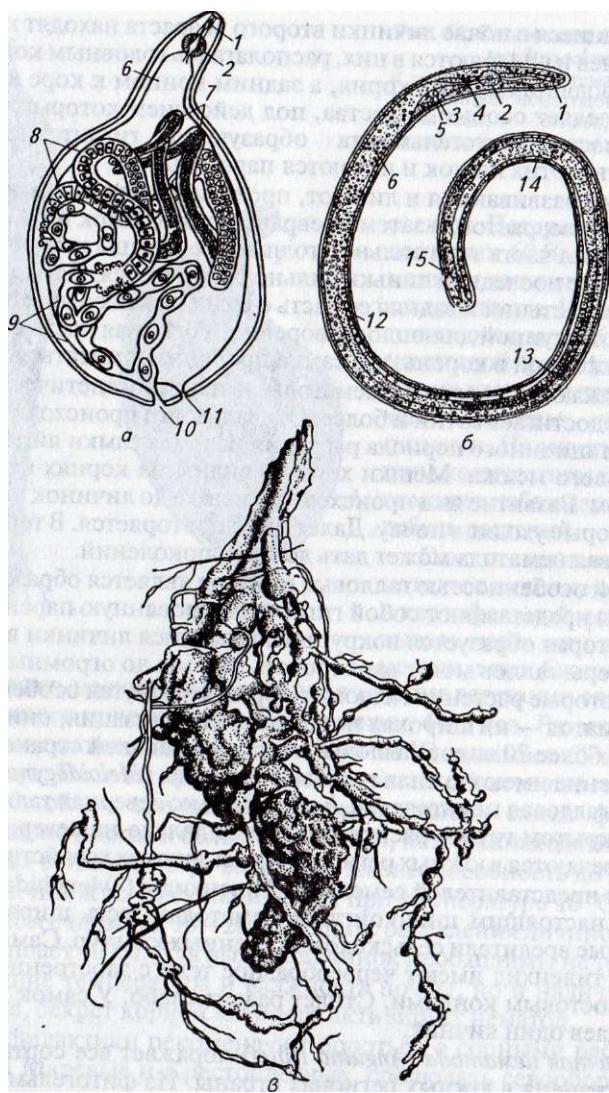


Рис. 65. Галловая нематода:

6 - самец в галле на корнях огурца; 1 - стилет; 2 - бульбус пишева
 7 - яичник Леза; 3 - Невное кольцо; 3' - "Делительное отверстие"; 6 - Гшка
 Г" тм' * - яйцевод; 9 - матка с яйцами; 10 - половое отверстие; 11 - анус
 ~ семяприемник; 13 - семяпровод; 14 — семязвергательный канал; 15 - спикулы



Рис. 66. Пшеничная нематода:
а — самка; б — молодое растение пшеницы, зараженное нематодой; в - • галлы; г — здо-
ровый колос пшеницы; д — колос, пораженный нематодой

Жизненный цикл пшеничной нематоды приспособлен к циклу развития растения-хозяина. Весной вместе с галлами фитогельминты попадают в почву с высеваемым зерном, в почве могут находиться и галлы, которые осипались с поврежденных колосьев осенью. Внутри галлов содержатся личинки второго возраста. При разбухании галлов под действием почвенной влаги личинки выходят и двигаются по направлению к проросткам пшеницы, но не далее 15—20 см от материнского галла. Значительная часть личинок погибает. Достигшие проростков пшеницы личинки заползают в пазухи листьев и становятся эктопаразитами. Яровые посевы поражаются весной, а озимые могут повреждаться осенью.

В период формирования колоса фитогельминты перебираются в зачатки цветков злаков, из которых формируются галлы. Для окончательного формирования галла необходимо присутствие нескольких самцов и самок нематоды. В галле они достигают половой зрелости и копулируют. Самка откладывает до 2,5 тыс. яиц. Из яиц появляются личинки, которые питаются ослизненной тканью галла. В одном галле может обитать до 17 тыс. личинок. Галлы завершают свое развитие несколько позже зерен пшеницы. В это время личинки заканчивают питаться и впадают в анабиоз. Взрослые особи погибают. Часть зрелых галлов выпадает из колосьев еще до уборки урожая, но большая их часть высвобождается из колосьев во время обмолота пшеницы, засоряя зерно. В состоянии анабиоза в галле личинки могут сохранять жизнеспособность в течение 25 лет. Пора-

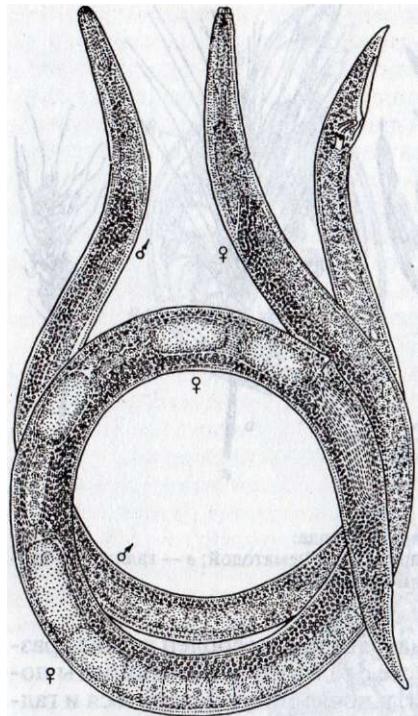


Рис. 67. Стеблевая нематода картофеля

на, затем кожура сморщивается и растрескивается, открывая доступ гнилостным микроорганизмам. В итоге происходит разрушение клубней.

Весь цикл развития стеблевой нематоды картофеля происходит внутри тканей растения-хозяина. Основной источник расселения фитогельминта — поврежденный посадочный материал, что определяет меры профилактики и борьбы с этим фитогельминтом.

Нематоды из большого маточного клубня по мере роста растения переходят по столонам в молодые клубни. Реже заселение молодых клубней происходит из почвы. В стеблях и клубнях паразиты размножаются, давая за лето несколько поколений. Осенью основная масса нематод остается в клубнях, а часть мигрирует в почву. Особое внимание следует уделять уборке послеуборочных остатков и их утилизации.

Представителем семейства настоящих шишкоиглых нематод является *стеблевая нематода на луке и чесноке (Ditylenchus dipsaci)*. Это многоядный фитогельминт, у которого насчитывают до 20 экологических рас, различающихся по тому, на каких растениях эти расы паразитируют. Луковая

жленные растения пшеницы отстают в росте, стебель искривляется.

Главная мера борьбы с пшеничной нематодой — предпосевная очистка зерна пшеницы, использование в севооборотах для очистки полей от нематод чистых паров и т. п.

Стеблевая нематода картофеля (Ditylenchus destructor) является представителем семейства настоящих шишкоиглых нематод (Tylenchidae) и встречается на всей территории РФ. Нематода имеет небольшие размеры: длина тела у самок 0,7—1,4 мм, у самцов — 0,7—1,3 мм. Стилет небольшой и тонкий. Стеблевая нематода поражает в основном картофель и вызывает разрушение клубней по типу сухой гнили (рис. 67). Однако могут повреждаться также морковь, горох, томат и некоторые сорные растения. Признаки поражения в виде небольших беловатых пятен видны после срезания кожуры клубня картофеля. В период хранения на пораженных клубнях появляются серые пят-

раса широко распространена в нашей стране. По строению стеблевые нематоды на луке и чесноке сходны со стеблевой нематодой картофеля.

У поврежденных луковиц репчатого лука рыхлые чешуи, их ткань набухает. У лука-севка на боковой стороне поврежденной луковицы заметно ярко-белое пятно — очаг размножения фитогельминта. Заселенные растения отстают в росте, а сильное поражение вызывает гибель посевов. После гибели растения нематоды уходят в почву. Стеблевая нематода может впадать в анабиоз и сохранять жизнеспособность в течение 2—3 лет.

В благоприятных условиях нематоды размножаются в тканях растений непрерывно, давая поколение за поколением. Нижний порог активности фитогельминта лежит в пределах 7—9 °С. После уборки урожая стеблевая нематода сохраняется в послеуборочных остатках поврежденных растений, реже в почве. Если лук хранят при высоких температурах, то пораженные луковицы усыхают, а если при низких температурах, то загнивают. Основными источниками распространения луковой расы стеблевой нематоды являются лук-севок и почва, что и определяет профилактику и борьбу с этим фитогельминтом.

В нашей стране широко распространены и другие расы стеблевой нематоды, в том числе на землянике (*D. dipsaci*), которая по своему строению идентична луковой расе. У пораженных растений видны вздутия на стеблях, черешках и листьях. Эти утолщения, где происходит размножение фитогельминта, становятся мягкими и трухлявыми. Размножаются и развиваются эти нематоды только в наземных активно растущих частях без выхода во внешнюю среду. Зимует нематода в сердечках земляники, реже в листьях. За сезон она дает до четырех-пяти поколений. Главной причиной распространения нематоды является использование заселенной ею рассады земляники.

Особенности экологии фитонематод. Жизненный цикл фитонематод связан с растениями и почвой. Взаимоотношения нематод с окружающей внешней средой, и прежде всего с растениями, определяют целый ряд важных особенностей, которыми обладают эти животные, свидетельствующих о путях развития круглых червей от типичных почвенных форм до узкоспециализированных вредителей растений. На основе взаимоотношения с растением построена экологическая классификация нематод.

Типичные сапробионты — нематоды, живущие в гниющей среде. Часть из них обитают в мертвой древесине, другие — в разлагающихся корнях, третий — в погибших зеленых растениях. У типичных сапробионтов кутикула имеет повышенный коэффициент полупроницаемости, что препятствует проникновению вредных веществ в тело нематод из гнилостной среды. Следует учитывать, что сапробиотические процессы сопровождаются постоянной сменой бактериальной флоры, что естественно ведет к соответствующим изменениям и в видовом составе нематод.

Нетипичные сапробионты. Для этих нематод также характерно обитание в гнилостной среде, но они могут заселять и живую ткань растений, проникая в нее самостоятельно или вслед за патогенными организмами. У таких нематод повысилась проницаемость кутикулы, изменя-

нился характер движения и замедлились процессы развития и темпы размножения. Нематоды в значительной степени зависят от бактерий и их деятельности, так как сами черви не способны воздействовать химическим путем на растительную ткань. Они способны лишь к повреждению растений, измельчению тканей и проглатыванию ее мелких частиц.

Прикорневые нематоды обитают в ризосфере и связаны с растением. Эти нематоды либо имеют стилет, либо являются хищниками. Первые, прокалывая стилетом ткани растения, питаются главным образом его соками, вторые пытаются животными организмами, обитающими в прикорневой части растений.

Фитогельминты относятся к настоящим вредителям растений. Для них характерны постоянное пребывание в органах растений и развитие в пищеводе желез, секрет которых способен разрушать ткани растения-хозяина. При этом изменения происходят не только в поврежденном органе, но и в растении в целом.

Все фитогельминты вооружены стилетом, а их кутикула обладает наивысшими барьерными свойствами. Фитогельминты не совместимы с сапробиотической средой: при загнивании пораженного органа они уступают место сапробиотическим организмам, лишь некоторые фитогельминты могут существовать в гниющих остатках, питаясь, по-видимому, гифами грибов. Таким образом, некроз ткани приводит к переселению фитогельминта в здоровые части растения или в почву.

Воздействие нематод на растения. Многие фитогельминты приспособились к жизни в ограниченном круге растений-хозяев. Например, стеблевая нематода картофеля живет только в тканях растений, которые богаты углеводами. Фитогельминт вырабатывает большое количество амилолитических ферментов, гидролизующих крахмал клубней до Сахаров. Отсюда и разрушение клубня по типу сухой гнили вследствие оттягивания воды из поврежденной части клубня в здоровую.

Стеблевая нематода на луке и чесноке вызывает сильное набухание поврежденного участка луковицы из-за роста активности пектиназы, разрушающей чешую.

Галловые нематоды вызывают изменения в процессах обмена веществ растения. Фитогельминты ведут неподвижный образ жизни на корнях растений, поэтому поврежденные корни не разрушаются, а только сильно видоизменяются. Вокруг переднего конца гельминта формируется группа гигантских многоядерных клеток, а в паренхиме корня происходит усиленное разрастание ткани, приводящее к образованию галла. Нематода питается за счет гигантских клеток, в которые паразит вводит стилет и выделяет пищеварительные ферменты. Клетка, выделяя вещества-ингибиторы, быстро тормозит действие гидролитических ферментов нематоды. В результате нематода получает часть пищи и ткань растения остается живой.

Фитогельминты хорошо приспособились к жизни во всех органах растений-хозяев. Патогенные фитогельминты иногда могут быть полезными, если поражают сорные растения. В перспективе возможно будет использовать некоторые виды нематод в борьбе с сорняками.

Не только фитогельминты воздействуют на растения, но и растения могут оказывать существенное влияние на вредителей. Известно, например, что при выращивании растений под пленочными укрытиями, а не в теплицах из стекла значительно снижается ущерб от галловых нематод. Оказывается, что пленочное укрытие пропускает больше ультрафиолетовых лучей, которые способствуют усиленному синтезу в растении ингибиторов, подавляющих ферменты фитонематод.

Через почву и растения-хозяев на фитогельминтов опосредованно воздействует комплекс абиотических и биотических факторов. Из абиотических факторов наибольшее влияние оказывают температура, влажность, механический состав почвы, ее кислотность, обеспеченность кислородом и насыщенность солями.

Биотические факторы сложны и многочисленны: отношения нематод с микроорганизмами в агробиоценозе, внутривидовые отношения (между фитонематодами и свободноживущими нематодами, хищниками и паразитами, между отдельными видами фитонематод за пищу и т. п.), отношения между патогенными грибами и бактериями, естественные враги нематод и пр.

Абиотические факторы. Климатические факторы, в том числе температура и влажность, определяют географическое распространение нематод. Для каждого вида характерны свои оптимальные и предельные температуры (низкие и высокие), при которых происходит его развитие. Широкое использование приемов выращивания растений в закрытом грунте позволило фитогельминтам продвинуться далеко на север. Большинство видов цистообразующих нематод приспособлены к температурам и влажности средней полосы. Стеблевая нематода картофеля в малой степени зависит от влажности и поэтому распространена во всех картофелепроизводящих районах.

Для нематод, у которых почва является средой для развития или сохранения и активации личинок, решающее значение имеет сочетание оптимальных показателей влажности и температуры в период массового заселения надземной части и корней всходов инвазионными личинками.

У нематод, непрерывно развивающихся в тканях растений без выхода во внешнюю среду, имеется пик численности паразита, когда температура и влажность оптимальны для быстрого развития популяции.

Большое значение для нематод имеет механический состав почвы, поскольку по ее порам нематоды передвигаются. Поэтому большинство видов нематод обитают в легких почвах.

Биотические факторы. Наиболее изучены взаимосвязи нематод с различными организмами почвенного биоценоза. Некоторые из них являются естественными врагами нематод, другие (грибы и бактерии) выступают компонентами в сложных болезнях растений, третьи используют нематод в качестве переносчиков от одного растения к другому (вирусы, бактерии, грибы).

Естественными врагами нематод могут быть хищные нематоды, клещи, грибы, вирусы и бактерии. Последние вызывают болезни нематод и их гибель. Почвенные грибы как враги нематод имеются во всех

почвах, но особенно их много в почвах, богатых органикой. Серьезный ущерб нематодам наносят хищные грибы — гифомицеты. Они ловят своих жертв с помощью клейких колец, узлов или спор. Способ поражения хищными грибами простой: споры или кольца прорастают в тепло жертвы, и гриб начинает питаться нематодой. Есть грибы, поражающие яйца в цистах нематод. В таких пораженных яйцах личинки погибают. Грибы из родов *Verticillium* и *Nematophthora*, поражающие самок цистообразующих нематод, в Великобритании предлагают использовать для борьбы с овсяной нематодой.

К врагам нематод относятся амебы, инфузории, споровики, тихоходки, членистоногие, в том числе клещи, и хищные нематоды.

К неблагоприятным условиям среды (низкой температуре зимой, сухости почвы, отсутствию растений-хозяев) нематоды приспособились по-разному. Самый распространенный способ — анабиоз (сохранение жизнеспособности в неблагоприятных условиях). В состояние анабиоза у большинства видов могут впадать личинки второго возраста, заключенные в цисты или галлы, а также личинки, находящиеся в растении, но в последнем случае анабиоз менее продолжителен.

Круглые черви — паразиты животных и человека. Значительное число круглых червей являются паразитами различных сельскохозяйственных животных, а также большинства диких позвоночных. Эти гельминты вызывают опасные заболевания, снижают продуктивность животных, причиняя существенный ущерб животноводству страны. Некоторые круглые черви паразитируют и у человека. Наиболее часто у животных и человека паразитируют нематоды из группы геогельминтов. Очень опасны для человека биогельминты трихинелла спиральная, нитчатка Банкрофта и другие. Рассмотрим биологические особенности наиболее распространенных в нашей стране и опасных паразитических видов нематод.

Аскариды (различные виды сем. *Ascaridae*) живут в кишечнике многих диких, домашних и сельскохозяйственных млекопитающих (свиньи, лошади, птица, кролики, мелкий и крупный рогатый скот, собаки и др.), а также человека, особенно часто детей. Особенностью аскарид является их видовая специфичность: каждому виду млекопитающего присущ свой вид аскариды (рис. 68). Свиная (*Ascaris suum*) и человеческая (*Ascaris lumbricoides*) аскариды весьма близки по многим признакам, но человек редко заражается свиной аскаридой, а свинья — человеческой. Одна из самых крупных аскарид паразитирует в тонком кишечнике лошадей (*Parascaris equorum*). У человека паразитирует аскарида, длина которой достигает 20 см.

Аскариды имеют веретенообразное тело длиной 20—40 см при диаметре до 3—5 мм. Хорошо развит половой диморфизм: самки значительно крупнее самцов, хвостовой отдел самцов загнут крючком. Плодовитость самок очень высокая — 200 тыс. яиц за сутки. Оплодотворение внутреннее. Оплодотворенное яйцо одевается прочными оболочками, которые хорошо защищают зародыш от неблагоприятных условий внешней среды. Погруженные в слабый раствор формалина

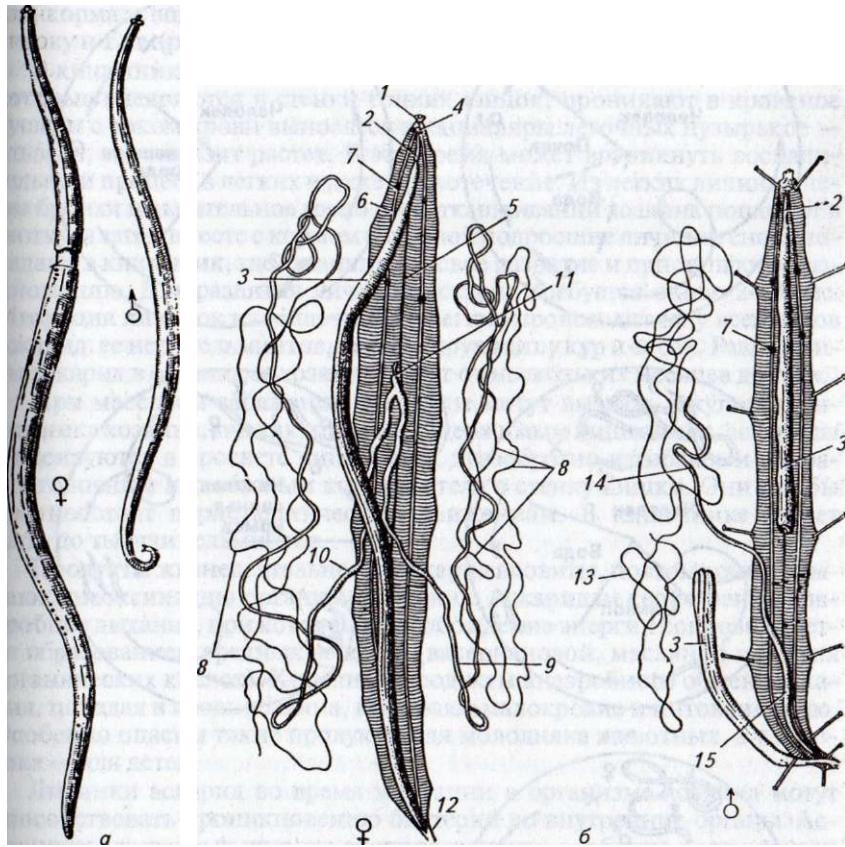


Рис. 68. Аскарида лошадиная:
 а — внешний вид; б — внутреннее строение: 1 — рот; 2 — пищевод; 3 — кишка; 4 — окологлоточное нервное кольцо; 5 — брюшной нервный тяж; 6 — боковой валик гиподермы с каналом выделительной системы; 7 — фагоцитарные клетки; 8 — яичники; 9 — яйцеводы; 10 — матка; // — влагалище; 12 — анальное отверстие; 13 — семенник; 14 — семязапровод; 15 — семязвергательный канал

яйца аскарид сохраняют жизнеспособность более месяца. Вредно действуют на яйца ультрафиолетовые лучи.

Дробление яйца начинается в теле самки аскариды, но основное развитие личинки проходит во внешней среде в течение 8—30 сут, что определяется главным образом температурой среды. Во внешнюю среду яйца попадают с фекалиями хозяина. После окончания развития личинки яйцо становится инвазионным, т. е. способным к заражению хозяина. Проглатив такое яйцо, животное может заболеть аскаридозом. Заражение происходит при потреблении загрязненных яйцами аска-



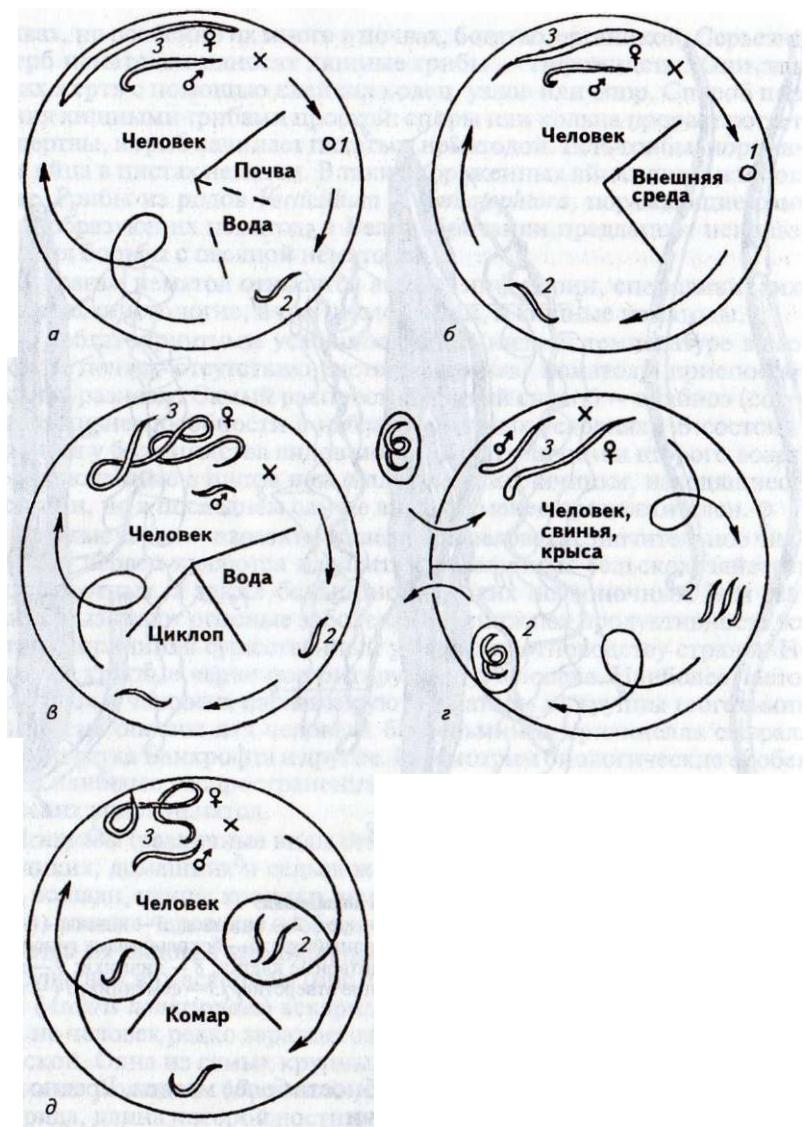


Рис. 69. Схемы жизненных циклов нематод — паразитов человека:
 а — геогельминты без смены хозяев и с миграцией личинок по крови (аскарида, власоглав и свайник); б — геогельминты без смены хозяев и без миграции личинок по крови (острица); в — геогельминты со сменой хозяев (ришта); г — биогельминты без смены хозяев (трихинелла); д — биогельминты со сменой хозяев (нитчатка Банкрофта); 1 — яйцо; 2 — личинка; 3 — половозрелые особи

рид корма и воды, а также через предметы ухода, оборудование, подстилку и т. п. (рис. 69).

В кишечнике хозяина из яиц выходят микроскопические личинки, которые внедряются в стенки тонких кишок, проникают в кровяное русло и с током крови выносятся в капилляры легочных пузырьков — ильвеол, где паразит растет. В это время может возникнуть воспалительный процесс в легких и даже кровотечение. Из легких личинки через бронхи и дыхательное горло при откашливании хозяина попадают в І лотку, а затем вместе с кормом и слюной подросшие личинки снова попадают в кишечник, где заканчивают свое развитие и приступают к размножению. Для развития личинок аскарид требуется около 2—3 мес. Миграция личинок из кишечника в легкие происходит не у всех видов аскарид: ее нет у гельминтов, паразитирующих у кур и собак. Разные виды аскарид в организме хозяина живут от нескольких месяцев до года.

При массовом заражении аскариды могут вызвать закупорку кишечника хозяина, так как, питаясь содержимым кишечника, аскариды фиксируются в просвете кишечника, дугобразно изгинаясь и упираясь головным и хвостовым концами тела в стенку кишки. Они как бы противостоят перистальтическим движениям. В кишечнике может быть до тысячи гельминтов.

Продукты жизнедеятельности аскарид ядовиты, поэтому они вызывают интоксикацию организма хозяина. Аскаридам свойственно анаэробное дыхание, при котором высвобождение энергии сопровождается образованием вредных веществ (валериановой, масляной и других органических кислот). Конечные продукты анаэробного обмена аскарид, попадая в кровь хозяина, вызывают малокровие и интоксикацию. Особенно опасны такие продукты для молодняка животных, а у человека — для детей.

Личинки аскарид во время миграции в организме хозяина могут способствовать проникновению бактерий во внутренние органы. Аскаридозы животных широко распространены, особенно часты случаи заболевания лошадей и свиней. В борьбе с аскаридозами важное значение имеют меры профилактики: чистота в помещении, правила гигиены, содержание животных (кроликов, птиц и др.) на сетчатых полах и др. Особое внимание уделяют удалению навоза, чистоте туалетов и борьбе с мухами — переносчиками яиц гельминтов.

Установлена роль земляных червей в распространении аскаридоза: черви заглатывают яйца аскарид вместе с растительными остатками и почвой. В кишечнике червей из яиц выходят личинки, которые сохраняют в червях жизнеспособность до года. Поедая дождевых червей, свиньи заражаются аскаридами. Подсосные пороссята заражаются через вымя. Личинки аскарид могут сохраняться в организме мух, личинках жуков и других насекомых, которые, как и дождевые черви, являются резервуарными хозяевами аскарид.

В последние годы получены данные, что при заглатывании человеком инвазионных яиц свиной аскариды личинки могут выходить из яиц и мигрировать по всему организму человека, оседая в различных

внутренних органах, например в мозге, печени и т. п. Это вызывает у человека тяжелое заболевание и общую интоксикацию организма.

В кишечнике кур паразитирует небольшая нематода, внешне похожая на аскариду, — *аскарида* (*Ascaridia galli*). Самки этой нематоды достигают в длину 11 см, самцы меньше — до 7 см. Яйца аскаридии после выхода с пометом наружу становятся инвазионными на 7—20-е сут. Куры и цыплята заражаются, поедая загрязненный инвазионными яйцами корм. В кишечнике птицы личинки высвобождаются из оболочек яиц, затем внедряются в стенки кишечника, где развиваются в течение 16—20 сут. Выйдя снова в просвет кишечника, личинки достигают зрелости. Весь цикл развития длится 35—58 сут. Особенностью развития аскаридий является отсутствие в их цикле миграции по телу хозяина.

Аскаридиоз вызывает массовую гибель молодняка, снижает яйценоскость взрослых кур. Резкому снижению заболеваемости аскаридиозом способствует содержание птицы в клетках или на сетчатых полах.

Острицы (различные виды сем. *Oxyuridae*) паразитируют в толстом отделе кишечника позвоночных и человека. Это мелкие паразиты, имеющие вздутие пищевода (бульбус) и тонкий заостренный хвостовой конец. Длина человеческой остирицы (*Enterobius vermicularis*) составляет менее 2 см, тогда как лошадиная остирица (*Oxyura equi*) достигает 6—18 см за счет своего длинного хвоста. Острицы *O. equi* вызывают болезнь оксиуроз у лошадей, мулов и зебр. Паразитируют оксиуры в большой ободочной кишке, но могут заселять слепую кишку и даже тонкий отдел кишечника. В результате у лошадей нарушается деятельность пищеварительного тракта и поражается кожа у корня хвоста. Паразиты распространены повсеместно.

После оплодотворения самки самцы погибают. Переполненные зрелыми яйцами самки вместе с фекалиями спускаются к анальному отверстию лошади. Они выходят из кишечника пассивно. Часть самок падают на землю и откладывают яйца на поверхность испражнений, а часть задерживаются в складках слизистой оболочки вокруг ануса и откладывают яйца в перианальной области под корнем хвоста лошади. Самки после откладывания яиц погибают. Клейкая слизистая масса, в которой находятся яйца, образует сероватый налет на перианальной области в ее складках.

Под хвостом и в области промежности через 2—3 сут яйца становятся инвазионными. При движении хвоста лошади яйца попадают во внешнюю среду, загрязняя подстилку, траву, стены денника, кормушки и т. п. Если инвазионные яйца попадут в кишечник лошади, то из них выходят микроскопические личинки и развиваются во взрослых гельминтов. Чаще оксиурозу подвержен молодняк и старые лошади. Больные животные испытывают сильный зуд в области хвоста, они расчесывают эти места о выступающие части денника, способствуя осипанию яиц.

У человека, чаще всего у детей, которые легко самозаражаются этим гельминтом, в толстой и задней кишке паразитирует детская остирица (*Enterobius vermicularis*). Это мелкие черви длиной 5—10 мм белого цвета. Оплодотворенные самки выползают ночью из прямой кишки и

откладывают яйца на кожу вокруг ануса. Уже через 10—12 ч яйца становятся инвазионными. Самка откладывает около 11 тыс. яиц. Яйца попадают на белье, постельные принадлежности, на пол и т. д. Из-за сильного зуда в области анального отверстия дети расчесывают это место и легко самозаражаются (автоинвазия).

Nematoda Toxascaris leonina из сем. Ascaridae во взрослой стадии паразитирует у молодняка старших возрастов и взрослых домашних и диких плотоядных животных, вызывая заболевание токсасскаридоз. Это черви светло-желтого цвета длиной 6—10 см (самки) и 4—6 см (самцы).

Выделенные с калом во внешнюю среду яйца при благоприятных условиях дозревают до инвазионной стадии в течение недели. У заглотивших их зверей в тонком отделе кишечника из яиц выходят личинки; они внедряются в стенку кишечника и линяют. Через некоторое время личинки возвращаются в просвет кишечника и через 3—4 нед достигают половой зрелости.

Токсасскаридозом не болеют новорожденные и молодые щенки, кроме щенков песца. Это болезнь взрослых плотоядных, в том числе кошек и собак.

У молодых щенков паразитирует *Toxocara canis*, а у кошек — *T. mystax*, вызывая заболевание, называемое токсокарозом. Токсокары (в отличие от токсасскарид, развивающихся прямым путем) в личиночной стадии мигрируют по организму (кишечник — кровяное русло — легкие — трахея — гортань — кишечник) в течение месяца.

Заражение плотоядных токсокарозом и токсасскаридозом возможно не только прямым путем, но и через резервуарных хозяев, которыми могут быть мыши и другие грызуны. Личинки *Toxascaris leonine* у грызунов концентрируются в стенках желудка или кишечника. У песцов наблюдается внутриутробный путь заражения животных.

Яйца токсокар и токсасскарид очень устойчивы к воздействиям неблагоприятных факторов среды: при обработке фекалий 5 %-ным раствором фенола они погибают только через 3 нед. При сильном заражении токсокары и токсасскариды оказывают токсическое действие, вызывая воспаление кишечника, а иногда и его закупорку. Из кишечника токсасскариды могут проникать в желчные протоки печени, протоки поджелудочной железы, желудок и даже трахею. Токсокары более опасны, так как часть мигрирующих с кровью личинок попадают в различные органы и ткани хозяина и там инкапсулируются, долго сохраняя жизнеспособность. Хищники, поедая животных, инвазированных цистами токсокар, заражаются ими. Есть токсокары, которыми заражаются дети, играя в песке на детских площадках, где испражняются больные кошки.

Стронгиляты (ряд видов подотряда Strongylata). Обширная и разнообразная группа круглых червей небольших размеров, паразитирующих в толстом отделе кишечника у разных позвоночных животных. В огромных количествах они встречаются и у сельскохозяйственных животных, в том числе лошадей и других непарнокопытных. Именно сельскохозяйственные животные в силу скученности их содержания и

ограниченности территорий пастбищ представляют для паразитических червей благоприятную среду для заселения.

Насчитывают около 45 видов нематод — возбудителей кишечных стронгилятозов лошадей. Все они относятся к геогельминтам и имеют сходное развитие во внешней среде. Кишечные стронгилятозы — самые распространенные и повсеместно встречающиеся гельминтозы. Почти все лошади с самого раннего возраста поражаются этими болезнями. Интенсивность заселения паразитами (от нескольких сотен особей до многих десятков тысяч) зависит от возраста, условий содержания и кормления животных. Часто стронгилятозы приносят существенный ущерб: от отставания в росте до летального исхода.

Паразиты имеют небольшие размеры: самцы достигают 0,5—4,5 см, самки крупнее. Оплодотворенные самки продуцируют множество яиц, которые с фекалиями попадают во внешнюю среду. При благоприятных условиях среды (8—38 °C) в яйце формируется личинка, которая выходит из яйца, развивается и линяет, достигая инвазионной стадии. При достаточной влажности личинки мигрируют горизонтально и вертикально в почве и по стеблям растений. Лошади заражаются, потребляя траву и воду, загрязненные инвазионными личинками. Заражение происходит в теплое время года на пастбище или в течение всего года в утепленных конюшнях.

В организме лошадей развитие различных стронгилят протекает неодинаково. У некоторых видов (деляфондии) личинки совершают миграцию: кишечник — кровеносные сосуды — тромбы в сосудах (где личинки развиваются) — кровяное русло — стенка кишечника — просвет кишечника. У других стронгилят (*Strongylus equities*) — самых крупных стронгилид лошадей (самцы достигают в длину 25—35 см, самки — 35—45 см), личинки через слизистую оболочку кишечника мигрируют в поджелудочную железу, где развиваются в течение 8 мес. Затем возвращаются в толстый кишечник. Весь срок развития составляет почти год.

В пищеварительном тракте жвачных животных паразитирует большое число видов нематод из подотряда Strongylata. Эти паразиты тоже относятся к геогельминтам. Развитие их во внешней среде протекает так же, как и у кишечных стронгилят лошадей.

Нематоды подотряда Strongylata вызывают также стронгилятозы органов дыхания у сельскохозяйственных животных. Заболевание диктиокаулез у жвачных животных вызывают нематоды рода *Dictyocaulus*. Это довольно крупные круглые черви с нитевидным телом белого цвета, паразитирующие в легких крупного рогатого скота, овец и других млекопитающих. Один из представителей, *Dictyocaulus viviparus*, вызывает опасное заболевание дыхательных путей у крупного рогатого скота. Эти нематоды паразитируют в бронхах и трахее животных. Болеет в основном молодняк крупного рогатого скота. Заболевание характеризуется развитием бронхита и бронхопневмонии.

Во внешней среде личинки становятся инвазионными через 4—10 сут после их выхода с фекалиями из организма хозяина. Личинки редко покидают фекалии. Их распространению способствует гриб

Pilobolus, который, раскрываясь, разбрасывает личинок на расстояние до 3 м. Распространению личинок способствуют также паводковые воды. С ними личинки попадают в водоемы. Особую опасность представляют мелкие лужи на пастбищах.

Телята, проглатившие личинок с травой или водой, заражаются дикиоикаулезом. Через стенку кишечника личинки с кровью мигрируют в легкие, где завершают свое развитие. Из легких половозрелые паразиты выбрасываются в кишечник.

В легких свиней паразитируют метастронгилиды из семейства *Metastrongylidae*, личинки которых живут в дождевых червях. Поедая дождевых червей, свиньи заражаются этими гельминтами, которые паразитируют в бронхах свиней. У поросят гибель может достигать 30 %.

Среди биогельминтов наибольшую опасность представляет *трихинелла спиральная* (*Trichinella spiralis*), жизненный цикл которой проходит полностью в организме хозяина. У трихинеллы различают две стадии: кишечные трихинеллы и мышечные трихинеллы. Хозяевами трихинелл могут быть хищники, парнокопытные, в том числе свиньи, насекомоядные, ластоногие, грызуны и человек. У человека эти гельминты вызывают заболевание трихинеллез. Человек заражается в основном от свиней и редко от других, в частности диких животных, потребляя мясо, пораженное мышечными трихинеллами (рис. 70). В мясе зараженных свиней рассеяны небольшие овальные капсулы. В каждой капсule находится скрученная в спираль микроскопическая трихинела длиной около 0,5 мм.

Если такое трихинеллезное мясо будет плохо термически обработано и съедено хозяином, то в его желудке под действием желудочного сока капсулы растворяются и молодые трихинеллы выходят из них. Попав в тонкий отдел кишечника, трихинеллы растут и через 2—3 сут превращаются в половозрелых гельминтов. Самки достигают в длину 3—4 мм, а самцы — 1,5 мм. Черви внедряются в ткань кишечника и приступают к размножению. После спаривания самцы погибают.

Оплодотворенные самки закрепляются головным отделом в

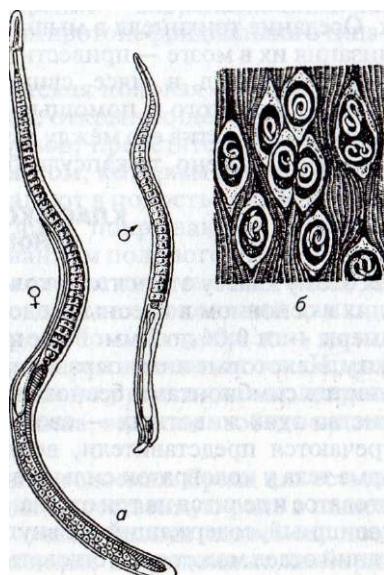


Рис. 70. Трихинелла *Trichinella spiralis*:
а - самка и самец; б—личинки, инкапсунированные в мышцах свиньи

слизистой оболочке кишки. Самки живут около 2 мес, и за это время каждая отрождает примерно 2 тыс. личинок (трихинеллам свойственно яйцеклеворождение). Личинки проникают в лимфатические сосуды стенки кишечника и затем в кровяное русло. С кровью личинки разносятся по всему организму и обычно попадают в мышцы. Личинки, активно двигаясь, внедряются в волокна поперечно-полосатой мускулатуры, где пытаются и растут, разрушая мышечные волокна. Затем трихинеллы закручиваются в спираль и постепенно окруждаются соединительной тканью капсулой. Примерно через год в стенках капсул откладывются соли углекислого кальция и капсулы принимают белый цвет. Так основной хозяин превращается в промежуточного.

Жизненный цикл трихинеллы может быть иным. Если сопротивляемость организма хозяина снижается, то отродившиеся личинки трихинеллы внедряются в ворсинки кишечника и, закончив там свое развитие, вновь возвращаются в просвет кишечника, где достигают половой зрелости. Это значительно увеличивает число паразитов в кишечнике и продлевает срок кишечной инвазии, а также усиливает поражение мышц.

Свиньи заражаются трихинеллезом, поедая зараженных трихинеллой дохлых крыс или свиные отходы с боен. Крысы же заражаются, питаюсь тканями павших от трихинеллеза других крыс или свиными отходами, попавшими на свалку или закопанными в землю на небольшую глубину. Зараженные трихинеллами туши свиней уничтожают, так как паразиты в капсулах чрезвычайно устойчивы к самым жестким режимам термической обработки мяса.

Для человека наиболее опасна мышечная стадия развития трихинеллы. Инкаспулирование личинок сопровождается болями в мышцах. Оседание трихинелл в мышцах глаз может вызвать слепоту, а локализация их в мозге — привести к смертельному исходу. Наличие личинок трихинелл в мясе свиней можно проверить в домашних условиях. Для этого с помощью острой бритвы делают тонкий срез мышцы, и, поместив его между двумя стеклами, рассматривают в лупу. Если мясо заражено, то капсулы будут видны.

КЛАСС КОЛОВРАТКИ (*Rotatoria*)

К этому классу относятся около 1,5 тыс. видов круглых червей, живущих в основном в пресных водоемах и имеющих микроскопические размеры — от 0,04 до 2 мм. Реже их можно встретить в морях, болотах, во мху. Некоторые виды паразитируют у беспозвоночных, а некоторые являются симбионтами беспозвоночных. Однако подавляющее большинство этих животных — свободноживущие черви, хотя среди них встречаются представители, ведущие прикрепленный образ жизни. Форма тела у коловраток сильно варьирует, но у большинства оно продолговатое и делится на три отдела: головной с мерцательным аппаратом, туловищный, содержащий все внутренности, и задний, или ножной. Последний отдел может отсутствовать. Совокупность двух венчиков ресни-

Рис. 71. Схема строения коловратки:
 а — вид спереди; б — вид сбоку; 1 — гионные чувствительные щупальца; 2 — коловорачательный аппарат; 3 — рот; 4 — глотка с жевательным аппаратом; 5 — слюнные железы; 6 — пищевод; 7 — желудочные железы; 8 — желудок; 9 — яичник; 10 — протонефридий; 11 — задняя кишечка; 12 — мочевой пузырь; 13 — отверстие клоаки; 14 — семенные железы; 15 — ножной ганглий; 16 — пальцы ноги; 17 — клоака; 18 — спинные чувствительные щупальца; 19 — надглоточный ганглий

чек образует коловорачательный аппарат, вызывающий водоворот, который направляет мелкие частицы ко рту коловратки (рис. 71).

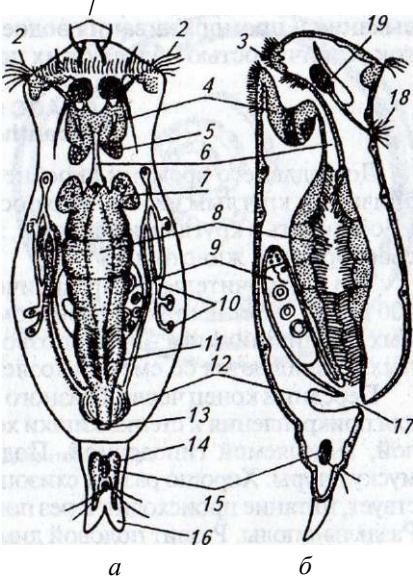
С помощью ноги коловратки, недущие прикрепленный образ жизни, временно или постоянно прикрепляются к субстрату. В ноге хорошо развиты мышцы. Кожно-мускульный мешок отсутствует.

Пищеварительная система представлена ртом, глоткой, пищеводом, объемистым желудком и кишкой, которая заканчивается анальным отверстием. Глотка имеет расширение — зоб, снабженный хитиновыми челюстями. Органы выделения протонефридиального типа открываются в клоаку.

Коловратки раздельнополы. Женская половая система состоит из яичника, желточника и яйцевода, открывающегося в клоаку. У самцов имеется семенник, семяпровод, предстательные железы и копулятивный орган с острым стилетом, которым самец прокалывает покровы самки. Спермии попадают в полость тела и проникают в яичник. Оплодотворенные яйца покрываются скорлупой. Размножение происходит с чередованием полового и партеногенетического поколений. В последнем случае получаются только самки. После нескольких партеногенетических поколений самки дают яйца с гаплоидным набором хромосом: из таких яиц выходят карликовые самцы и самки. Оплодотворенные самки откладывают покоящиеся яйца. Самцы обнаружены не у всех видов коловраток.

Из яиц сидячих форм выходит личинка, ведущая свободноплавающий образ жизни.

Коловратки многочисленны в наших водоемах. Среди них есть донные и планктонные формы, играющие существенную роль в пищевых цепях водоемов. Они же являются очистителями водоемов, поедая большое количество бактерий. Почвенные и придонные коловратки



способны к анабиозу, перенося таким образом длительное время пересыхания и промораживания водоемов. У планктонных видов коловраток устойчивостью обладают их покоящиеся яйца.

КЛАСС СКРЕБНИ (*Acanthocephala*)

До недавнего времени скребней выделяли в самостоятельный тип, близкий к круглым червям. Но последние исследования подтверждают их общность с круглыми червями, хотя морфологически скребни очень своеобразные животные.

Это исключительно паразитические черви, насчитывающие около 500 видов, обитающих во взрослом состоянии в кишечнике позвоночных, а в личиночном — у беспозвоночных животных, рыб и земноводных. Развиваются со сменой хозяев.

Передний конец червеобразного тела превращен в хоботок с крючьями для прикрепления к стенке кишки хозяина. Тело покрыто нежной кутикулой, выделяемой гиподермой. Под гиподермой располагается два слоя мускулатуры. Хорошо развит схизоцель. Пищеварительная система отсутствует, питание происходит через покровы тела. Органы чувств не развиты. Раздельнополы. Развит половой диморфизм. Развитие с метаморфозом.

Гигантский скребень (*Macrocanthorhynchus hirudinaceus*) длиной 25—60 см паразитирует в кишечнике свиней. Яйца попадают во внешнюю среду. Для дальнейшего развития яйца должны быть проглочены личинками бронзовок, майских жуков и др. Личинки жуков и взрослые жуки, зараженные личинками скребня, съедаются свиньями при выгульном их содержании. Сильное заражение свиней может привести к гибели животных.

Известно много скребней, которые во взрослом состоянии обитают в кишечнике грызунов, водоплавающих птиц, рыб, тюленей и др. (рис. 72). У некоторых скребней число промежуточных хозяев может достигать трех.

Например, у скребней, паразитирующих в организме тюленей, два промежуточных хозяина — мелкие ракообразные и рыба.

ФИЛОГЕНИЯ ПЕРВИЧНОПОЛОСТНЫХ ЧЕРВЕЙ

Считается, что первичнополосные круглые черви ведут свое начало от турбелляриеподобных предков. В разных классах круглых червей сохранились признаки, общие с плоскими червями: участки ресничного эпителия, протонефридии, участки паренхимы в схизоцеле. Наиболее близки к предкам современные коловратки и брюхоресничные. Схема экологической радиации круглых червей представлена на рис. 73.

ТИП НЕМЕРТИНЫ (*Nemertini*)

Немертины — свободноживущие морские черви, реже паразитические и пресноводные, предками которых предположительно были свободноживущие плоские черви. Известно около 750 видов немертин,

Рис. 72. Скрепни:
а — великан; б — четковидный

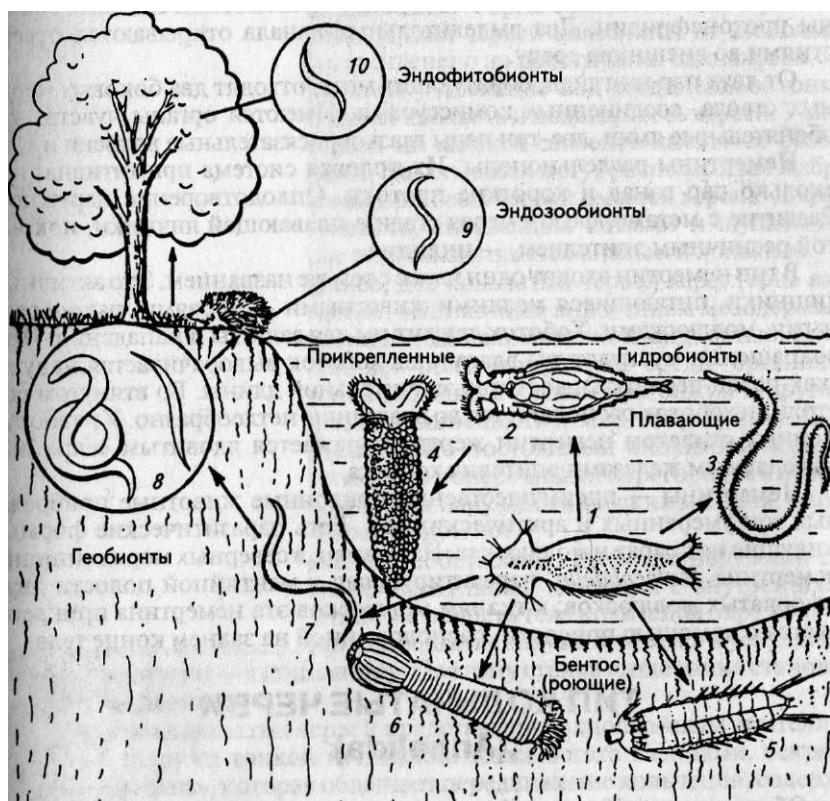


Рис. 73. Экологическая радиация Nemathelminthes:
1 — брюхоресничный червь; 2 — коловратка; 3 — волосатик; 4 — яйцо; 5 — киноринха;
6 — приапулида; 7 — нематода водная; 8 — почвенные нематоды; 9,10 — паразитические
нематоды

обитающих преимущественно в прибрежной зоне морей. Червеобразное тело (до 2 м) покрыто мерцательным эпителием (как у турбеллярий), под которым находятся кольцевые и продольные слои мышц (кожно-мускульный мешок). На переднем конце тела имеется выворачивающийся хоботок, зачастую вооруженный стилетом. С его помощью немертины захватывают добычу, которую затем отправляют в щелевидный рот, расположенный сentralной стороны у основания хоботка.

Полость тела отсутствует, промежутки между органами заполнены паренхимой, лишь влагалище хоботка выстлано целомическим эпителием. Пищеварительная система состоит из передней, средней и задней кишок. Аналльное отверстие располагается на заднем конце тела. Средняя кишка имеет боковые карманы.

У немертин появляется развитая кровеносная система, состоящая из трех основных сосудов: спинного и двух боковых, связанных между собой кольцевыми сосудами. С сосудами кровеносной системы связаны протонефридии. Два выделительных канала открываются отверстиями во внешнюю среду.

От двух пар ганглиев, образующих мозг, отходят два боковых нервных ствола, соединенных комиссарами. Имеются органы чувств; это обонятельные ямки, две-три пары глазков, осязательные волоски и т. п.

Немертины раздельнополы. Их половая система примитивна: несколько пар гонад и короткие протоки. Оплодотворение наружное. Развитие с метаморфозом через стадию плавающей личинки, покрытой ресничным эпителием, — пилидия.

В тип немертин входит один класс с тем же названием. Это активные хищники, питающиеся мелкими животными — червями, ракообразными, моллюсками. Хоботок служит им для защиты и нападения. При сокращении мускулатуры влагалища хоботок выворачивается наружу (как палец перчатки), достигая значительной длины. Во втянутом состоянии хоботок располагается во влагалище петлеобразно. У невооруженных стилетом немертин жертва поражается ядовитым секретом, выделяемым железами эпителия хоботка.

Немертины — преимущественно придонные животные прибрежных зон умеренных и арктических вод. Есть паразитические формы, живущие на крабах и моллюсках. Например, в северных морях типична немертина *Malacobdella*, паразитирующая в мантийной полости двустворчатых моллюсков; к тканям моллюсков эта немертина прикрепляется с помощью присоски, расположенной на заднем конце тела.

ТИП КОЛЬЧАТЫЕ ЧЕРВИ (*Annelida*)

Общая характеристика. Кольчатые черви, или кольчечцы, — наиболее высокоорганизованная группа червей с усложненными по сравнению с другими типами червей системами органов. Это двустороннесимметричные животные, характеризующиеся наличием у них вторич-

Июй полости тела, или целома. Метамерия у кольчатых червей выражается в том, что их тело снаружи расчленено на ряд однотипных сегментов, в каждом из которых повторяются многие органы. Известно около 12 тыс. видов кольчатых червей, ведущих свободный образ жизни главным образом в морях, а также в пресных водоемах и в почве. Многие из них имеют важное значение, так как служат кормом для бес-
I юзовоночных и позвоночных животных, участвуют в процессах почво-образования, служат объектом разведения, используются в медицинских целях и т. д. Паразитических видов немного. Кольчцы активно участвуют в деструкции органического вещества, вовлекая тем самым высвободившиеся биогенные элементы в круговороты. Особенно многообразны морские формы, которые живут на разных глубинах до 10 км.

Тип Кольчатые черви включает три класса: Многощетинковые черви (Polychaeta), Малощетинковые черви (Oligochaeta) и Пиявки (Hirudinea).

Строение и жизненные отправления. Форма тела кольчатых червей вытянутая и слегка уплощенная. Длина червей колеблется от нескольких миллиметров до метра. Тело расчленено на практически одинаковые сегменты (гомономная сегментация), имеющие вид соединенных тонкой кожей колец, что обеспечивает гибкость и подвижность червей. У многощетинковых морских червей на каждом сегменте находятся особые парные выросты — параподии. На сегментах могут располагаться жабры. У большинства видов многощетинковых червей имеется хорошо дифференцированная головная лопасть, снабженная глазами и шупальцами (рис. 74). Сегментированное тело заканчивается анальной лопастью.

Как уже указывалось ранее, для кольчатых червей характерна *вторичная полость тела*, или *целом*, выстланный эндотелием мезодермального происхождения и заполненный целомической жидкостью. Целомическая жидкость выполняет роль внутренней среды организма, функции гидроскелета, в ней находятся клетки (фагоциты и другие), она участвует в переносе различных веществ и защите организма. В целоме поддерживается относительно постоянный биохимический режим. Целом разделен посегментно поперечными перегородками на камеры: каждый сегмент имеет свою пару целомических мешков; у многих представителей этих перегородок нет.

Первичная полость не имеет собственных стенок: с наружной стороны ее ограничивает кожно-мускульный мешок, а с внутренней — стенка кишечника. Вторичная же полость тела кольчецов окружена однослоистым эпителием, прилегающим снаружи к кожно-мускульному мешку, а внутри — к кишечнику. Поэтому стенка кишечника становится как бы двойной.

Покровы кольчатых червей представлены однослойным эпителием, одетым снаружи тонкой кутикулой. Кожа богата железами, секретирующими слизь, которая облегчает передвижение кольчецов, содержит половые атTRACTАНты, ядовитые вещества и т. п. У морских представителей эти секреты используются при постройке домиков.

Нервная система развита лучше, чем у других червей. Она представлена парными спинными мозговыми ганглиями и брюшной нервной

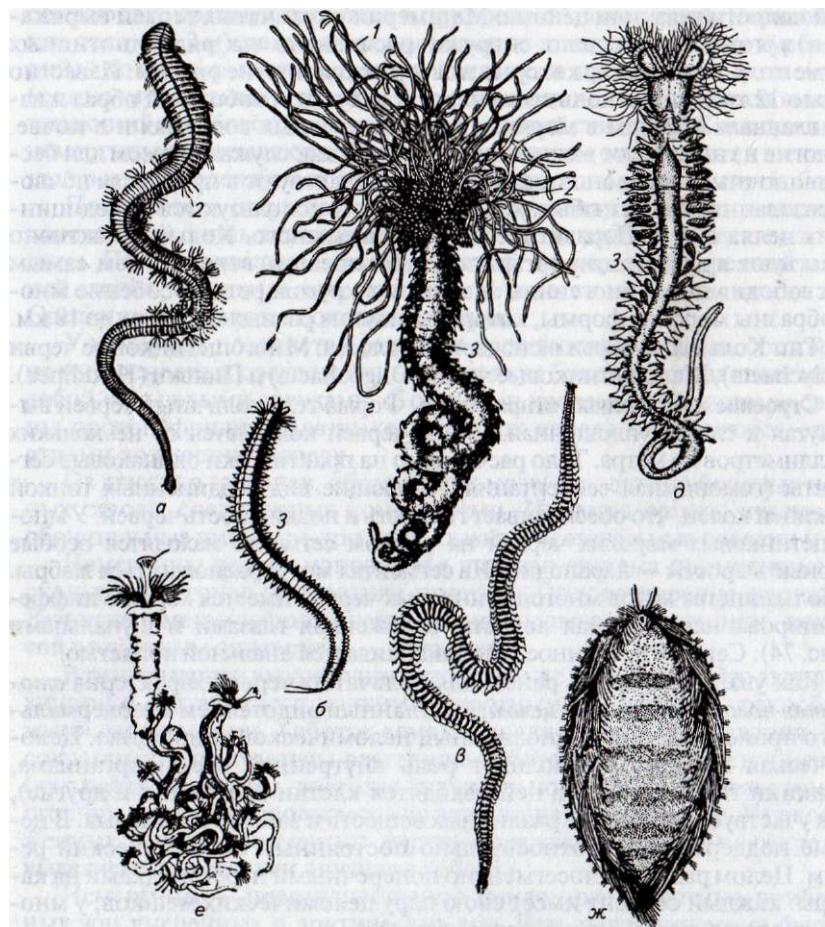


Рис. 74. Виды многощетинковых кольчатах червей:
а—пескожил (*Arenicola*); б—нереис (*Nereis*); в—филлодоце; г—амфитrite (*Amphitrite*);
д—сабеллария (*Sabellaria*); е—серпула (*Serpula*); ж—морская мышь (*Aphrodite*); 1—шу-
пальца; 2—жабры; 3—параподии

цепочкой с метамерно повторяющимися парными ганглиями в каждом сегменте. От ганглиев отходят нервы к различным органам. Нервная система типичных кольчатах червей развита значительно лучше и устроена сложнее (рис. 75). Появление головного мозга, расположенного дорсально над глоткой, существенно отличает кольчатах червей от плоских. Парные спинные доли мозга кольчецов разделены на передний, средний и задний ганглии.

Органы чувств у почвенных червей представлены многочисленными чувствующими клетками в кожном покрове. У морских многощетинковых

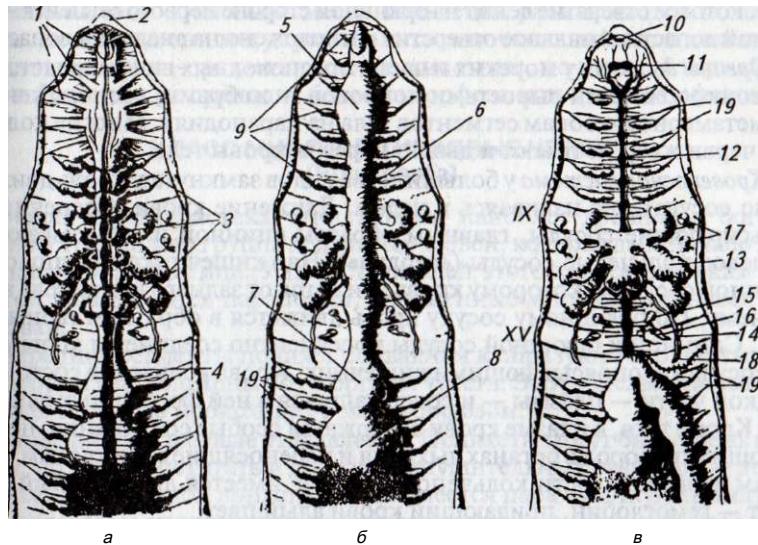


Рис. 75. Передняя часть тела дождевого черва:
и — червь с обнаженными спинным и кольцевыми кровеносными сосудами; б — пищеварительная система; в — нервная и половая системы; / — связки между глоткой и стенкой тела; 2 — ротовая лопасть; 3 — кольцевые сосуды («сердца»); 4 — спинной кровеносный сосуд; 5 — глотка; 6 — пищевод; 7 — зоб; 8 — мускульный желудок; 9 — метанефриди; 10 — рот; 11 — надглоточный ганглий и окологлоточное нервное кольцо; 12 — брюшная нервная цепочка; 13 — семенные мешки; 14 — семяпровод; 15 — яичник; 16 — воронка яйцевода; 17 — семяприемники; 18 — отверстие семяпровода; 19 — перегородки между сегментами; IX—XV — сегменты тела

червей хорошо развиты глаза и щупальца. Кожа кольчецов чувствительна к световым раздражителям. У некоторых форм имеются органы равновесия.

Мускулатура входит в состав хорошо развитого кожно-мышечного мешка; снаружи лежат кольцевые мышечные волокна, а внутри — продольные. У некоторых форм между этими слоями мышечных волокон может лежать третий слой диагональных мышечных волокон. С их помощью черви могут не только изгибать тело в разных направлениях, но и укорачивать или удлинять его. Мускулы входят в структуру части внутренних органов. Движению кольчецов способствуют особые придатки: щетинки и параподии. Щетинки имеют хитиновую природу (они приводятся в действие специальными пучками мышц).

Пищеварительная система. Кишечник состоит из трех отделов; первый и последний отделы выстланы эпителием эктодermalного происхождения, а средний отдел — энтодермального происхождения. Благодаря кровеносной системе процессы переваривания и всасывания питательных веществ идут более активно, что улучшает обеспечение органов и тканей. У части видов средняя кишка имеет глубокое втячивание (тифлозоль), увеличивающее поверхность кишечника. Имеется рот и анальное отвер-

стие. Ротовое отверстие лежит на брюшной стороне первого сегмента — головной лопасти. Анальное отверстие расположено на анальной лопасти.

Органы дыхания у морских и части пресноводных видов представлены тонкостенными выростами покровов — жабрами, расположеными метамерно по бокам сегментов тела на параподиях. Многие кольчатые черви жабр не имеют и дышат через покровы тела.

Кровеносная система у большинства видов замкнутая: кровь движется по сосудам, не изливаясь в целом. Движение крови обеспечивают пульсирующие сосуды, главным образом спинной и опоясывающие пищевод кольцевые сосуды («сердца»). Над кишечником расположен спинной сосуд, по которому кровь движется от заднего конца тела к переднему. По брюшному сосуду кровь движется в обратном направлении. Спинной и брюшной сосуды посегментно соединены кольцевыми сосудами, опоясывающими кишечник. Кровь кольчецов состоит из жидкой части — плазмы — и содержащихся в ней форменных элементов. Кроме того, в плазме крови содержатся особые соединения, поглощающие кислород в органах дыхания и переносящие его к тканям и органам червей. У части кольчецов в плазме имеется дыхательный пигмент — гемоглобин, придающий крови алый цвет.

Органами выделения служат посегментные метанефридиальные эктодермальные образования. Обычно в каждом сегменте имеется одна пара нефридиев. Каждая пара метанефридиев начинается в одном сегменте воронками с ресничками, открытыми в целом, от которых выделительные каналы продолжаются в следующем сегменте и открываются там наружу парными отверстиями. Метанефридии — это не только органы выделения, но и органы регулирования водного баланса в организме червей. В каналах метанефридиев происходит концентрация конечных продуктов обмена (аммиак превращается в мочевую кислоту), а освободившаяся вода снова поступает в целомическую жидкость. Это особенно важно для почвенных и наземных кольчецов (рис. 76). Помимо метанефридиев на стенах целома разбросаны специальные (хлорагенные) клетки, которые поглощают из целомической жидкости конечные продукты обмена. Много таких клеток и на стенах средней кишки. Нагруженные конечными продуктами обмена хлорагенные клетки могут выводиться через метанефридии наружу.

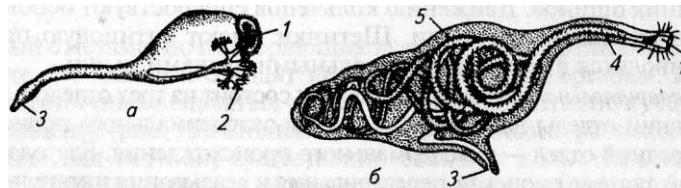


Рис. 76. Строение метанефридия кольчецов:
а — метанефридий с пламенными клетками; б — типичный метанефридий: 1 — воронка;
2 — пламенные клетки; 3 — наружное отверстие; 4 — канал органа; 5 — кровеносные со-
суды, разветвляющиеся в стенах канала

Размножение. Основная масса кольчецов размножается половым путем, но иногда они могут размножаться почкованием или делением. Кольчатые черви — раздельнополые животные, но много и гермафродитов. Развитие их проходит без метаморфоза или с метаморфозом.

КЛАСС МНОГОЩЕТИНКОВЫЕ ЧЕРВИ (*Polychaeta*)

Общая характеристика. Это главная, наиболее древняя и богатая (около 8 тыс. видов) группа кольчатых червей, которая дала начало другим классам этого типа. У представителей этого класса по бокам сегментов тела имеются параподии, снабженные многочисленными щетинками.

Передние сегменты полихет сливаются и образуют головной отдел, на котором расположены рот и органы чувств. Это раздельнополые животные. Развитие происходит с метаморфозом.

Строение и жизненные отравления. Полихеты могут быть очень мелкими, но могут достигать и довольно внушительных размеров — до 1 м и более. На головной лопасти всегда имеется пара чувствующих щупиков, которые у сидячих форм превратились в крону щупальцевидных пришатков. На головной лопасти расположена пара осязательных щупалец — антенн. Форма тела вытянутая, туловище состоит из разного числа сегментов — от 5 до 800.

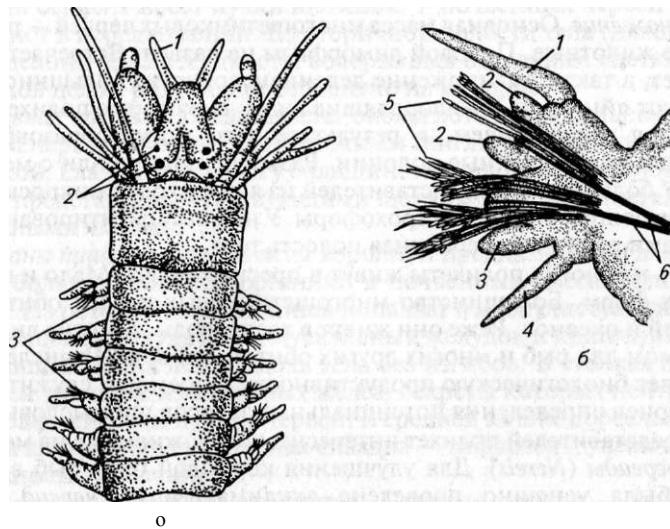


Рис. 77. Голова и параподия многощетинкового кольчатого червя нереис:
а — голова: 1 — щупальца; 2 — глаза; 3 — параподии; б — параподия: 1 — спинной усик;
2 — спинная лопасть; 3 — брюшная лопасть; 4 — брюшной усик; 5 — щетинки; 6 — опорные щетинки

Передвигаются полихеты с помощью параподий (рис. 77), образованных основной нерасчлененной частью и двумя лопастями: спинной и брюшной. От основания спинной и брюшной лопастей каждой параподии отходит по тонкому усiku, выполняющему обонятельные и осязательные функции. Параподии соединены с телом червя подвижно и действуют по типу простого рычага. В каждой лопасти есть пучок упругих и тонких щетинок. Кроме того, в толще основной части имеется пара опорных щетинок, более толстых и длинных. С помощью параподий черви ползают по дну водоемов, у плавающих форм параподии выполняют функцию плавников. У некоторых полихет параподии частично или полностью редуцированы (роющие виды; виды, живущие в домиках).

Покровы у полихет, живущих на дне водоемов, представлены хорошо развитой кутикулой. У активно плавающих форм, живущих в домиках или зарывающихся в грунт, кутикула тонкая. Органы чувств хорошо развиты: на голове одна-две пары глаз, осязательные усики, обонятельные ямки и щупальца. У примитивных форм эпителий местами может быть ресничным.

Органы дыхания. Дышат полихеты жабрами или поверхностью тела. У большинства функцию дыхания берут на себя участки параподий.

Кровеносная система представлена спинным и брюшным сосудами, а также кольцевыми сосудами. Кровеносная система замкнута. Движение крови по телу обеспечивается сокращениями главным образом спинного сосуда. Кровь может быть окрашена в красный цвет.

Размножение. Основная масса многощетинковых червей — раздельнополые животные. Половой диморфизм не развит. Встречается партеногенез, а также размножение делением поперек. Большинство откладывают яйца, есть и живородящие виды. Некоторые полихеты размножаются почкованием, в результате чего могут образовываться временные разветвленные колонии. Развитие прямое или с метаморфозом. У большинства представителей из яиц выходят микроскопические плавающие личинки — трохофоры. У них несегментированное тело с рядами ресничек, первичная полость тела.

Очень немногие полихеты живут в пресных водах. Мало и паразитических форм. Большинство многощетинковых червей обитают на дне морей и океанов. Реже они живут в толще воды. Многие виды служат кормом для рыб и многих других обитателей вод. Их численность определяет биологическую продуктивность водоемов и служит одним из критериев определения потенциальных запасов промысловых рыб.

Из представителей полихет интересны черви, живущие на мелководье, — *нереиды* (*Nereis*). Для улучшения кормовой базы рыб в нашей стране была успешно проведена акклиматизация *нереид* (*Nereis diversicolor*) в Каспийском море, куда их завезли из Азовского. Многощетинковый червь *пескожил* (*Arenicola marina*) во множестве заселяет песчаные отмели, живя в заиленном песке и питаясь органикой; подобно дождевому черви он пропускает грунт через свой пищеварительный тракт.

У тихоокеанского многощетинкового червя *пололо* (*Eunice viridis*) период размножения сегменты задней части тела заполняются половыми продуктами. Затем эти части тела отрываются и всплывают на поверхность океана. Яйца и спермин выходят из разрывов сегментов в воду, где и происходит оплодотворение. Из зигот образуются плавающие личинки, которые развиваются во взрослых червей и опускаются на дно.

КЛАСС МАЛОЩЕТИНКОВЫЕ ЧЕРВИ (*Oligochaeta*)

Общая характеристика. К этому классу принадлежат многие водные и почвенные формы, в том числе дождевые черви. Известно более 5 тыс. видов малощетинковых червей, из которых в морских водах встречается очень небольшое их число. У малощетинковых червей нет параподий. По бокам тела у них имеются небольшие пучки щетинок (щетинок нет на передних и задних сегментах). У дождевых червей на каждом сегменте имеется по четыре пары щетинок. Головной отдел не обособлен. У большинства нет щупалец. Гермафродиты. Развитие прямое.

Строение и жизненные отправления. Цилиндрическое и сильно вытянутое тело малощетинковых червей состоит из похожих друг на друга сегментов, число которых может колебаться от 5 до 600. На переднем конце расположено ротовое отверстие, на заднем — анальное. Некоторые тропические виды могут достигать 2 м и более.

В покровах червей расположено много желез, выделяющих слизь. Каждый сегмент несет пучки щетинок. У почвенных червей щетинки участвуют в передвижении. Во вторичной полости тела (целоме) находится целомическая жидкость, содержащая отдельные клетки. У многих видов целом разделен посегментно на камеры.

Нервная система представлена окологлоточным кольцом, надглоточным нервным узлом, подглоточным ганглием и брюшной нервной цепочкой. Глаза и щупальца у большинства видов отсутствуют. Органы чувств представлены чувствующими щетинками, статоцистами и обонятельными ямками.

Органы пищеварения развиты хорошо и предназначены для пропускания больших объемов органики в почвенных массах или донного грунта. Из ротовой полости пища попадает в мускулистую глотку и затем в пищевод, оттуда в зоб, мускульный желудок и кишечник. Все органы пищеварения лежат вдоль тела без изгибов. В стенках пищевода имеются три пары известковых желез, секреты которых нейтрализуют гуминовые кислоты в пище червей. В средней кишке дорсально расположена внутренняя продольная складка — тифлозоль, увеличивающая поверхность кишечника.

Кровеносная система замкнутая. Главные сосуды — брюшной и спинной. В покровах малощетинковых червей имеется густая сеть капилляров, из которых обогащенная кислородом кровь собирается в сосуд, лежащий под брюшной нервной цепочкой. Таким образом, у малощетинковых червей, за редким исключением, дыхательная система отсутствует. В отличие от полихет у малощетинковых червей кольцевые

сосуды в области пищевода пульсируют и называются «сердцами». Кровь содержит гемоглобин, который растворен в плазме крови (у млекопитающих гемоглобин находится в эритроцитах).

Половая система. Малощетинковые черви гермафродиты, которым свойственно перекрестное оплодотворение, что и определяет сложность строения половой системы. Тело этих червей слегка уплощено и состоит из 50—250 внешне сходных сегментов. В области 32—37-го (от головной лопасти) сегментов имеется скопление одноклеточных железок, образующих кольцевидное утолщение — поясок (рис. 78). Эти клетки выделяют слизь для образования яйцевого кокона и белковую жидкость для питания зародышей.

В 10-м и 11-м сегментах тела дождевого червя располагается по паре семенников (см. рис. 75), которые прикрыты тремя парами семенных мешков. В семенных мешках созревают и накапливаются спермии. Из семенных мешков спермии поступают в мерцательные воронки семяпроводов. Семяпроводы сливаются попарно по левой и правой сторонам тела и образуют два продольных канала, открывающиеся на брюшной стороне двумя отверстиями на 15-м сегменте тела.

Рис. 78. Дождевой червь:
1 — женское половое отверстие; 2 — мужское половое отверстие; 3 — поясок

Женская половая система образована paarой мелких яичников, расположенных в 13-м сегменте. От яичников отходят два коротких яйцевода с воронками, которые открываются на брюшной стороне 14-го сегмента двумя половыми отверстиями. В 13-м сегменте яичники и воронки яйцеводов прикрываются яйцевыми мешками. Помимо этого к женской половой системе относятся две пары семяприемников, расположенные в 9-м и 10-м сегментах и открывающиеся на брюшной стороне двумя парами отверстий.

Спаривание у дождевых червей сводится к обмену спермой. В период размножения сначала все особи становятся самцами, поскольку у них развиты только семенники. Во время спаривания два червя двигаются головными концами друг к другу и соприкасаются брюшными сторонами, при этом поясок каждого червя располагается на уровне семяприемников, происходит обмен спермой. После этого черви расходятся. Затем у каждого из них на пояске, представляющем собой желобистое утолщение кожи нескольких определенных сегментов, образу-

ется муфта. Эта муфта сокращениями мускулатуры тела сдвигается к головному концу черва. Когда муфта проходит мимо 14-го сегмента, в нее откладываются яйца, а из 9—10-го сегментов туда попадают спермин. Происходит перекрестное оплодотворение. Наконец, муфта сбрасывается через головную часть черва, края ее смыкаются и она становится яйцевым коконом, в котором и происходит развитие зародышей. Кокон дождевых червей по форме напоминает лимон желто-бурого цвета. Диаметр кокона составляет около 4—5 мм.

Из яиц, развивающихся в коконе, выходит сформировавшийся червячок. У низших олигохет в коконе может быть несколько яиц. У высших, как правило, одно яйцо, несколько бывает редко. Помимо полового размножения у олигохет встречается и бесполое размножение: тело черва поперечно делится на две части, недостающие части регенерируют. У дождевых червей хорошо выражена способность к регенерации, причем легко восстанавливается задний конец тела, головной отдел восстанавливается редко.

Дождевые черви ведут активную и полезную для почвы деятельность. Их ходы способствуют аэрации почвы, по ним проникает вода, черви разрыхляют почву и удобряют ее остатками своей жизнедеятельности, богатыми гумусными кислотами. При благоприятных условиях на 1 м² площади луга обитают 50—100 дождевых червей. Существуют около 200 видов дождевых червей. Мелкие (менее 1 см) беловатые кольчатые черви семейства энхитреид (*Enchytreidae*) чаще всего встречаются в почве, но есть виды, обитающие в пресных водоемах. Почвенных энхитреид насчитывают около 400 видов. Плотность энхитреид в почве может составлять 150 тыс. на 1 м². Их легко разводить в искусственных условиях в качестве корма для рыб. Питаются эти черви органическими остатками.

Множество олигохет живет на дне водоемов, питаясь органикой. У некоторых видов водных малощетинковых червей наблюдается почкование. В этом случае образуются сложные цепи почкующихся осо-бей. Водные малощетинковые черви служат кормом для обитателей водоемов, а наземные — для многих наземных обитателей, в том числе для множества позвоночных.

Дождевые черви и биогумус. Дождевой червь ведет ночной образ жизни; в рыхлом грунте он делает норки, уплотняя землю. В плотном грунте черви вынужден делать норки путем выедания почвы. Часть ее, пропущенную через пищеварительный тракт, червь выбрасывает наружу. Разыскивая пищу, червь заднюю часть тела держит в норке. Найдя пищу (растительные остатки), затаскивает ее в норку и затем поедает. Черви не выносят сухости, ибо испытывают кислородное голодание. В проточной богатой кислородом воде могут жить несколько дней.

Идея промышленного культивирования дождевых червей принадлежит американскому врачу Барретту, который в 1947 г. опубликовал результаты своих опытов. В 1975 г. в Италии была создана промышленная технология. Только в США работают тысячи специализированных производств по переработке навоза в биогумус с помощью червей. Био-

гумус (переработанный дождевыми червями и другими организмами подстилочный навоз) — высокооценное органическое удобрение.

Поглощая вместе с почвой огромное количество распадающихся растительных остатков, микробов, грибов, водорослей, простейших, нематод и других почвенных организмов, дождевые черви переваривают их и выделяют с копролитами (копрос — испражнение, литое — камень) большое количество собственной кишечной микрофлоры, ферментов, витаминов и других биологически активных веществ. Биологически активные вещества обладают антимикробными свойствами, препятствующими развитию патогенной (болезнетворной) микрофлоры, гнилостных процессов, выделению зловонных газов, обеззаражающими почву и придающими ей приятный запах земли.

В процессе переваривания растительных остатков в пищеварительном тракте червей формируются *гумусные вещества*, которые отличаются по химическому составу от гумуса, образующегося в почве при участии только микроорганизмов. В кишечнике червей развиваются процессы полимеризации низкомолекулярных продуктов распада органических веществ и формируются молекулы *гуминовых кислот* и *фульвокислот* (последние неблагоприятны, и чем их меньше, тем ценнее гумус). Эти кислоты образуют с металлами соли — *гуматы* и *фульваты*.

Гуматы лития, калия и натрия растворимы и легко вымываются водой; они представляют собой наиболее ценную часть гумуса, которая легко доступна для растений. Гуматы кальция, магния, кремния и тяжелых металлов нерастворимы и составляют ту часть гумуса, которую можно назвать «консервами» почвенного плодородия. Они накапливались в черноземах весь послеледниковый период. Эти гуматы могут растворяться под влиянием ферментов корневой системы растений, но лишь в таких количествах, которые удовлетворяют их потребности. Эти соли не подвержены гидролизу, но оказывают большое влияние на создание ценной, прочной и пористой структуры, не подверженной влиянию эрозии.

Следует отметить, что гуматы тяжелых металлов еще более устойчивы к гидролизу ферментами корневой системы растений и практически не усваиваются ими. Это одно из главных свойств гумуса — связывание в почве тяжелых металлов и предохранение живого на Земле от их токсического действия, в том числе от действия тяжелых радионуклидов. Чем больше в почве гумуса, тем сильнее выражено это ее свойство. Поэтому пищевая продукция, выращенная на высокогумусных почвах, является экологически безопасной для животных и человека.

Деятельность дождевых червей замедляет вымывание из почвы подвижных питательных элементов, закрепляет тяжелые металлы и препятствует развитию водной и ветровой эрозии. В копролитах червей естественных популяций содержится 11—15 % гумуса в расчете на сухое вещество. В естественных местах обитания плотность популяции дождевых червей варьирует от 100 до 600 и более особей на 1 м². За летний период популяция из 50 червей в пахотном слое почвы на 1 м² прокладывает 1 км ходов и выделяет на поверхность копролиты слоем около 3 мм. Еще больше их остается в толще почвы. Каждый червь пропуска-

ст через пищеварительный тракт за сутки столько почвы, сколько весит он сам (средняя масса червя в почвенном слое около 0,5 г). В средней полосе активная деятельность червей продолжается 200 дней; за этот период, следовательно, каждый червь пропустит через себя около 100 г субстрата, т. е. на 1 га приходится 50 т переработанной почвы.

Особенно важным условием для жизни червей является достаточная влажность субстрата. Влажность почвы ниже 30—35 % тормозит развитие, а при влажности 20—22 % они погибают в течение недели. Максимальную массу и наибольшее число коконов получают при 70—85 % влажности субстрата (столько же воды содержится и в теле червя).

В кислой среде (рН 5) или сильно щелочной (рН более 9) черви погибают в течение недели. Оптимум для дождевых червей — нейтральная среда (рН около 7). Велика потребность червей в азоте: в богатой азотом почве их численность растет. Вот почему их много в навозе и на пастбище. Концентрация растворимых солей более 0,5 % смертельна для них. Такие соли, как углекислый кальций, углекислое железо, сернокислый алюминий и хлористое железо безвредны даже в больших количествах. Итак, оптимальными для развития дождевых червей условиями являются следующие: температура 15—22 °С, влажность — 60-75 % и рН 7,3-7,6.

Дождевые черви живут в верхних слоях почвы. Они не уходят в нижние слои на спячку до тех пор, пока земля не промерзнет на глубину 5—6 см и не появится снежный покров. При длительной оттепели черви могут выползать даже на снег. Обычно при 5 °С черви перестают питаться, освобождают кишечник и уползают в нижние слои почвы, где оцепеневают. Просыпаются они под воздействием вешних вод и теплого воздуха, проникающих к ним в норки.

Дождевые черви очень плодовиты. Откладка коконов происходит с весны до начала лета и затем осенью до ноября. За лето червь откладывает по 18—24 кокона, в каждом из которых по 1—24 яйца. Через 2—3 нед из яиц выплываются молодые особи, которые по прошествии 7—12 нед уже сами способны размножаться. Благоприятные условия жизни ускоряют половое созревание молодых червей. Взрослые черви могут жить до 10 и даже 15 лет, их длина достигает десятков сантиметров, а масса — до 10 г. Молодые черви при достижении половой зрелости весят до 1 г. Крупные черви являются самой ценной частью популяции, но из-за несовершенной технологии обработки почвы именно эта часть дождевых червей погибает в пахотном слое.

Калифорнийский червь был выведен в 1959 г. в США. Этот червь отличается от своих диких сородичей тем, что обладает способностью размножаться в наземных культуваторах без всяких построек или теплиц. Калифорнийский червь в теплом климате дает 16—18-кратное воспроизводство за цикл культивирования под открытым небом и 512-кратное в условиях закрытых теплиц (дикие сородичи дают лишь 4—6-кратное воспроизводство). Калифорнийский червь стал предметом экспорт-импорта вместе с технологией его культивирования.

В нашей стране первые работы по культивированию калифорнийского червя начались лишь в 1984 г. под руководством А. М. Игонина. Им были разработаны звенья технологического процесса производства биогумуса с помощью калифорнийского червя, а также, что может быть самым главным, с использованием местных дождевых (навозных) червей. После переработки 1 т компостированного навоза получают 0,5 т гумусного удобрения 50 %-ной влажности и 6—10 кг живых червей.

КЛАСС ПИЯВКИ (*Hirudinea*)

Общая характеристика. Пиявкам свойственно своеобразное кольчатое строение. Их тело уплощено и не имеет четко выраженного головного отдела. Наружная кольчатость пиявок не соответствует более крупной внутренней сегментации тела. Сегментация тела однородная (гомономная). Каждому истинному сегменту соответствует 3—5 наружных колец. Тело пиявок состоит из 30—33 сегментов. Это придает им большую гибкость и позволяет вести активный образ жизни. Щетинки на теле отсутствуют. У большинства представителей этого класса имеются присоски: передняя и задняя. Передняя присоска окружает рот. Анальное отверстие находится над задней присоской.

Известно около 400 видов пиявок, живущих в основном в пресных водоемах и являющихся эктопаразитами беспозвоночных и позвоночных животных. Большинство пиявок питаются кровью разных животных, но на своих жертвах остаются недолго. В основном же паразиты ведут свободный образ жизни. Много пиявок, которые не сосут кровь, а являются хищниками. Есть сухопутные виды, распространенные в тропиках. В тропических лесах живут древесные и почвенные пиявки, нападающие на теплокровных животных и человека. В фауне нашей страны встречается 50 видов пресноводных пиявок.

Строение и жизненные отправления. Строение пиявок отвечает полу-паразитическому образу жизни, который они ведут (рис. 79).

Покровы представлены плотной кутикулой. Под ней лежит богатый железистыми клетками однослойный эпителий, образующий кутикулу. У основания эпителиального слоя разбросаны пигментные клетки, придающие пиявкам соответствующую окраску.

Мускулатура развита очень хорошо: кожно-мускульный мешок состоит из трех слоев мышечных волокон. Полость тела частично редуцирована и представлена системой лакун, заполненных паренхимой. Плотные покровы и паренхима защищают тело от высыхания и позволяют пиявкам длительное время пребывать на сушке, совершая миграции.

Нервная система. У пиявок имеется брюшная нервная цепочка. Глаза, если они есть, примитивны; в покровах располагаются чувствующие клетки и нервные окончания.

Дышат пиявки через покровы тела, но у некоторых видов имеются жабры.

Пищеварительная система хорошо приспособлена к образу жизни пиявок как паразитов. В ротовой полости у части видов имеются три

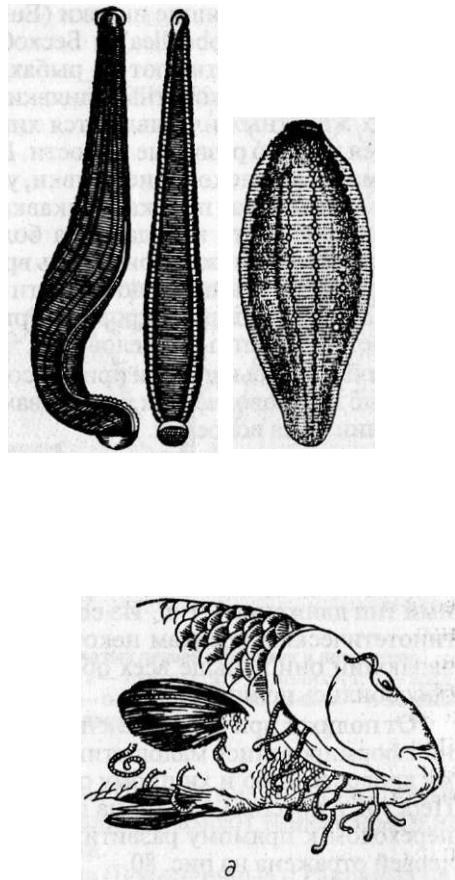
Рис. 79. Виды пиявок:
a, б — медицинская пиявка *Hirudo medicinalis* (со спинной и брюшной стороны); *в* — клепсина; *г* — ложноконская пиявка; *д* — рыбья пиявка *Piscicola geometra*

челюсти со множеством зубчиков (челюстные пиявки), у других пиявок есть хоботок, с помощью которого черви внедряются в покровы своих жертв (хоботные пиявки). Глотка пиявок выполняет функции сосущего аппарата, в который открываются протоки слюнных желез. У кровососущих пиявок (медицинская пиявка) в слюне содержится белковое вещество, препятствующее свертыванию крови в кишечнике червей, — гирудин. Передняя кишка имеет карманообразные боковые выросты, позволяющие сделать значительный запас крови: медицинская пиявка делает запас на 2—3 мес жизни, поскольку кровь долго сохраняется под действием гирудина в свежем виде. Процессы переваривания происходят в энтодермальном среднем отделе кишечника, который превращается в объемистый желудок с карманами.

Органы выделения — метанефриции.

Кровеносная система развита только у низших пиявок и частично у хоботных. У челюстных пиявок кровеносная система редуцируется, а ее роль выполняет лакунарная система целомического происхождения.

Половая система. Пиявки гермафродиты. Размножаются только половым путем весной около водоемов в сырых местах. Оплодотворение внутреннее и перекрестное. У пиявок, как и у малощетинковых червей, на 9—11-м сегментах находятся поясок и железы, выделяющие слизистые муфты для образования коконов. Коконы, похожие на мелкие желеуди, пиявки откладывают на землю. Развитие прямое и в коконе длится около 5 нед. Живут пиявки до 20 лет.



Подкласс Настоящие пиявки (*Euchirudinea*) делится на два отряда: Хоботные (*Rhynchobdellea*) и Бесхоботные (*Arhynchobdellea*). Хоботные пиявки паразитируют на рыбах, птицах, лягушках, моллюсках и ракообразных. Бесхоботные пиявки паразитируют только на позвоночных животных или являются хищниками. У них нет хоботка, но имеются хорошо развитые челюсти. Наиболее часто встречаются большая и малая ложноконские пиявки, улитковая пиявка, в южных регионах — медицинская пиявка, в Закавказье — конская пиявка. В природе пиявки чаще всего нападают на больных и ослабленных животных. Конская пиявка может причинить вред лошадям и скоту: во время водопоя из естественных водоемов эти пиявки проникают в носоглотку, гортань и могут вызвать кровопотери и удушье. Медицинская пиявка нападает на животных и человека.

Птицы и рыбы пиявки при массовом заселении могут вызывать гибель рыб в рыбоводческих хозяйствах и водоплавающей птицы при содержании ее на водоемах.

ФИЛОГЕНИЯ И ЭКОЛОГИЧЕСКАЯ РАДИАЦИЯ КОЛЬЧАТЫХ ЧЕРВЕЙ

В разных классах кольчатых червей есть признаки, которые свидетельствуют о родстве аннелид с низшими червями: первичная полость тела, протонефридии у части кольчатых червей и их личинок, ресничный тип движения и т. п. Из современных аннелид наиболее близки к гипотетическим предкам некоторые архианнелиды. На раннем этапе эволюции они раньше всех обособились от других аннелид. Позднее обособились полихеты.

От полихет при переходе к пресноводному и наземному образу жизни сформировались малощетинковые черви. При переходе к активному кровососанию и хищному образу жизни обособился класс пиявок. Переход в пресные воды и на сушу пиявок и олигохет сопровождался переходом к прямому развитию. Экологическая радиация кольчатых червей отражена на рис. 80.

ТИП МОЛЛЮСКИ (*Mollusca*)

Общая характеристика. Моллюски, или мягкотельые, — вторичнополостные животные с несегментированным телом, в большинстве случаев заключенным в раковину. Известно более 113 тыс. видов моллюсков, живущих в морских и пресных водах и на суше. Большинство водных моллюсков — обитатели дна. Это животные, ведущие начало от кольчатых червей. Тело моллюсков состоит из трех отделов: головы, туловища и ноги. Важнейшей их особенностью является наличие мантии — складки кожи, свешивающейся со спины, с разнообразными и многочисленными железами, которые вырабатывают секреты, используемые при построении раковины. Между телом и мантией образуется мантийная

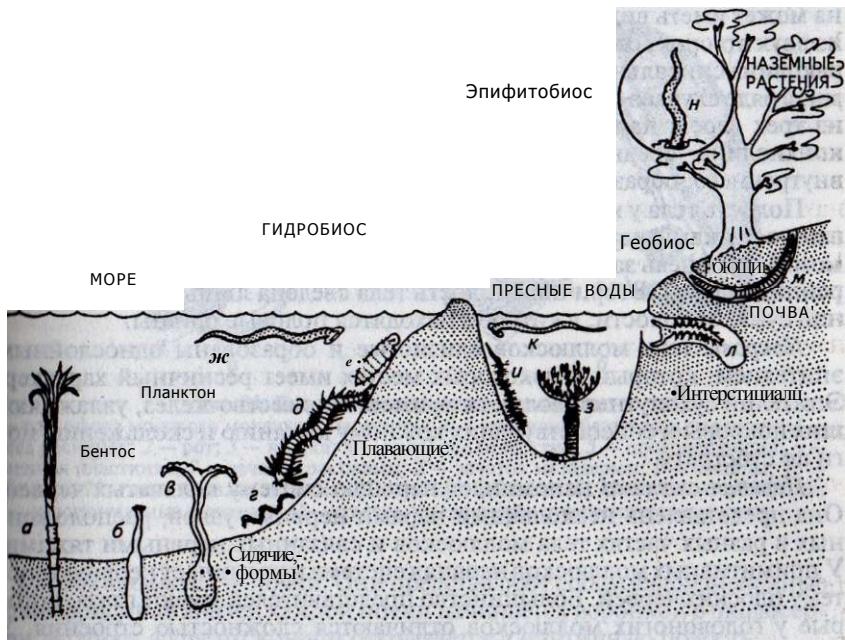


Рис. 80. Морфоэкологическая эволюция кольчатых червей и близких к ним типов:
а — погонофора; б — силинулида; в — бонелия; г — олигохета; д — полихета; е — архиннелида;
ж — пиявка морская; з — полихета; и — олигохета; к — пиявка; л — пещерная полихета;
м — дождевой червь; и — древесная пиявка

полость, в которой расположены жабры (или легкие — у сухопутных форм) и некоторые органы чувств. Моллюски имеют тонкие и мягкие покровы, богатые слизистыми железами.

Тип Моллюски включает два подтипа (Боконервные и Раковинные) и несколько классов, из которых наиболее распространены и представляют интерес три класса: Двустворчатые моллюски (Bivalvia), Брюхоногие моллюски (Gastropoda) и Головоногие моллюски (Cephalopoda), относящиеся к подтипу Раковинные (Conchifera).

Строение и жизненные направления. Размеры и форма тела моллюсков весьма разнообразны, что связано с особенностями их сред обитания. У части моллюсков тело имеет двустороннюю симметрию, но у многих, заключенных в спиральную раковину, тело асимметрично. Большая часть внутренних органов расположена в туловище. На голове находятся органы чувств, а внутри заключены крупные головные нервные узлы. У многих моллюсков голова не обособлена от тела. Нога служит для передвижения животного.

У большинства моллюсков имеется известковая раковина, образующаяся за счет выделений мантии. Раковина выполняет защитные функции, к ней прикрепляются мускулы и некоторые органы. Ракови-

на может иметь вид колпачка; у некоторых моллюсков она образована из двух створок, соединенных зубчатым замком или связкой. У других раковина спирально закручена или представлена несколькими щитками; в ряде случаев она может подвергаться редукции. Раковина состоит из трех слоев: наружного, построенного из органического вещества конхиолина, среднего, сложенного из известковых образований, и внутреннего, образованного тонким слоем перламутра.

Полость тела у моллюсков смешанная — *миксоцель*, так как образована остатками первичной полости и сильно редуцированным целомом. Миксоцель заполнен паренхимой, в которой расположены внутренние органы. Вторичная полость тела сведена лишь к околосердечной сумке и полости, в которой находятся половые органы.

Покровы тела моллюсков слизистые и образованы однослойным эпителием, который в некоторых местах имеет ресничный характер. Эпителий сухопутных моллюсков имеет множество желез, увлажняющих покровы и способствующих кожному дыханию и скольжению ноги по субстрату.

Нервная система напоминает нервную систему кольчатых червей. Она представлена несколькими парами нервных узлов, расположенных в разных частях тела моллюсков и связанных нервными тяжами. У примитивных моллюсков нервная система напоминает нервную систему плоских червей. Большинство представителей имеют глаза, которые у головоногих моллюсков отличаются сложностью строения. У мягкотелых есть органы осязания, органы химического чувства и равновесия (статоцисты).

Мускулатура слагается из гладких мышечных волокон, что обуславливает замедленное движение тела. У ведущих активный образ жизни головоногих моллюсков есть поперечно-полосатые мышцы, особенно хорошо развитые в ноге.

Органы дыхания у большинства водных видов представлены жабрами, расположенными в мантийной полости. Сами жабры — это видоизмененные участки мантии. Жабры имеют вид лепестков, расположенных на оси жабры. Все сухопутные и вторичноводные формы дышат легкими, которые также образованы участками мантии, и снабжены большим количеством кровеносных сосудов.

Кровеносная система незамкнутая. В околосердечной сумке (целоме) расположено сердце, имеющее желудочек и одно или несколько предсердий. От желудочка отходят артерии, разносящие кровь по всему телу. Кровь изливается в систему лакун полости тела. Отсюда кровь заасасывается в венозные сосуды и по ним поступает в жабры или легкое. Окисленная кровь возвращается в сердце по сосудам.

Органы пищеварения представлены ротовым отверстием, глоткой и пищеводом, из которого пища попадает в желудок. Для большинства видов характерно наличие в глотке аппарата для размельчения пищи — терки; нередко могут быть развиты и хитиновые челюсти. За желудком начинается кишечник, в который впадает проток печени. Кишечник оканчивается анальным отверстием.

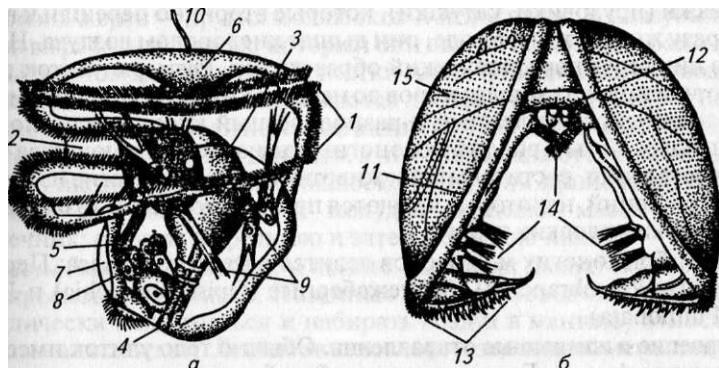


Рис. 81. Личинки двустворчатых моллюсков:
и — трохофора морского моллюска-парусника; б — глохидий беззубок и перловиц; 1 — нога с ресничками; 2 — рот; 3 — кишка; 4 — анальное отверстие; 5 — протонефридий; 6 — теменная пластинка; 7 — зародыш раковины; 8 — зародыш мезодермы; 9 — мышцы; 10 — реснички; 11 — личиночная раковина; 12 — мускул-замыкатель; 13 — зубцы с зубчиками; 14 — чувствительные щетинки; 15 — железы, выделяющие прикрепительные нити

Органы выделения — почки. Они представляют собой видоизмененные метанефриции. Каналец каждой почки начинается воронкой в околосердечной сумке, а другим концом открывается в мантийную полость.

Органы размножения находятся в целоме. Протоки половых желез открываются либо в мантийную полость, либо в протоки почек.

Большинство моллюсков раздельнополы. Оплодотворение яйце-клеток осуществляется в мантийной полости или внутри материнского организма. Развитие моллюсков происходит с метаморфозом или без него. Живущие в воде моллюски откладывают яйца в воду, а сухопутные формы — в почву и на ее поверхность. Есть живородящие виды. У низших форм из яйца развивается трохофорная личинка. У болыпинства моллюсков из яйца выходит личинка парусник, имеющая пучок ресничек и парус с лопастями, несущими реснички. Парус служит для движения личинки (рис. 81). У части морских, у большинства пресноводных и у всех сухопутных моллюсков развитие прямое.

КЛАСС БРЮХОНОГИЕ МОЛЛЮСКИ (Gastropoda)

Общая характеристика. У брюхоногих моллюсков, или улиток, нога имеет широкую подошву; такая нога используется этими моллюсками для ползания. При наличии раковины последняя спирально завита, что придает телу моллюска асимметричную форму. Голова имеет одну-две пары щупалец, у многих хорошо развиты глаза. Дышат брюхоногие моллюски жабрами или легкими. Первично брюхоногие моллюски — обитатели моря, но некоторые из них в процессе эволюции приспособились к жизни в пресных водоемах и на суше. Имеются пресноводные

моллюски (прудовики, катушки), которые вторично перешли к водному образу жизни: живя в воде, они дышат кислородом воздуха. Немногие из них ведут паразитический образ жизни. Размеры улиток варьируют от нескольких миллиметров до нескольких десятков сантиметров. Это самый многочисленный и разнообразный класс моллюсков. Известно более 90 тыс. видов брюхоногих моллюсков, питающихся гниющими остатками, растительной и животной пищей. Многие из них вредители растений, некоторые являются промежуточными хозяевами паразитических плоских червей.

Класс брюхоногих моллюсков делится на три подкласса: Переднекаберные (*Prosobranchia*), Заднекаберные (*Opisthobranchia*) и Легочные (*Pulmonata*).

Строение и жизненные отправления. Обычно тело улиток имеет разнообразную форму. Голова хорошо обособлена от туловища, которое образует сверху вырост в виде внутренностного мешка, закрученного спирально. Одна-две пары щупалец на голове несут осознательные функции и способны втягиваться. Хорошо развиты глаза, которые могут быть расположены на вершинах щупалец. Есть органы химического чувства и равновесия — статоцисты, расположенные в ноге. Передвигаются брюхоногие моллюски скольжением за счет волнобразных изгибов широкой и плоской подошвы ноги, представляющей мускулистый брюшной вырост.

Цельные, обычно спирально закрученные раковины имеют разнообразные форму и расцветку. У части плавающих брюхоногих раковина может быть рециклирована в той или иной степени. Нет ее и у наземных слизней, ведущих сумеречный и ночной образ жизни. В раковине размещено только туловище моллюска, но при опасности в раковину втягивается все тело. Раковина состоит из наружного тонкого органического слоя, под которым расположен минеральный слой известковой природы. Углекислая известь извлекается моллюсками из воды и пищи. Вещество для раковины выделяется известковыми железами мантии. У некоторых моллюсков раковина имеет еще и третий слой — внутренний перламутровый, или эмалевый, разной окраски. Мантийная полость расположена в нижних витках раковины. В мантийную полость открываются анальное отверстие, мочеточники, иногда и проток половых органов. У водных видов в мантийной полости расположены жабры. У наземных и вторичноводных моллюсков мантийная полость стала легким, которое открывается наружу специальным дыхательным отверстием.

Органы пищеварения. Брюхоногие моллюски питаются растительной пищей, детритом или являются хищниками. Ротовое отверстие расположено на нижней стороне головы и ведет в глотку, которая имеет роговые челюсти и мускулистый валик — язык с теркой (радулой), имеющей вид пластинки с мелкими зубчиками. С ее помощью моллюск отделяет частицы пищи или субстрата. В глотку впадают протоки слюнных желез. У некоторых хищных форм в слюне содержится серная кислота, с помощью которой моллюски растворяют раковины или пан-

цири своих жертв — других моллюсков и иглокожих. У ядовитых моллюсков вырабатывается яд, который они вводят особыми зубами в тело жертвы и потом поедают ее. Из глотки пища попадает в пищевод и затем в желудок, в который открываются протоки печени. Секрет печени характеризуется амилолитической активностью. Печень способна всасывать часть питательных веществ, в ней откладываются резервные липиды и гликоген. У низших моллюсков в печени происходит внутриклеточное пищеварение. Из желудка пищевая масса поступает в кишечник: сначала в среднюю и затем в заднюю кишку.

Органы дыхания — жабры и легкие. Жабрами дышат все морские и часть пресноводных видов. Наземные и вторичноводные (вынуждены периодически подниматься и набирать воздух в мантию) относятся к легочным моллюскам. Легкие моллюсков — это видоизмененная мантийная полость, стенки которой пронизаны сетью кровеносных сосудов. Воздух поступает в легкое через особое дыхальце, которое при погружении в воду закрывается. У моллюсков имеется и кожное дыхание.

Кровеносная система незамкнутая; ее образуют сердце, находящееся в околосердечной сумке, сосуды и лакуны. Кровь обычно бесцветна и содержит амебоциты. В околосердечную сумку открываются воронки двух почек; мочеточники выводят мочу в мантийную полость сбоку от анального отверстия.

Органы размножения. Большинство морских брюхоногих моллюсков раздельнополы, а наземные и многие пресноводные — гермафродиты. Половые железы непарные. Оплодотворение яйцеклеток осуществляется в материнском организме. Часто оплодотворенные яйца окружаются оболочками или студенистыми коконами и соединяются в кладки. У гермафродитов половая система устроена сложно. Так, у виноградной улитки имеется гермафродитная железа,рабатывающая яйцеклетки и спермии. Спаривание сводится к обмену спермой (подобно дождевым червям), которая поступает в семяприемники другой особи. Поэтому оплодотворение у них всегда перекрестное. Развитие протекает без стадии личинки (прямое) или с метаморфозом, реже наблюдается живорождение.

Брюхоногие моллюски участвуют в круговороте веществ в водоемах и на суше, потребляя органические остатки и перерабатывая их. Многие моллюски служат кормом для водных позвоночных. Морские трубачи являются источником черного и розового жемчуга, высоко ценящегося на мировом рынке.

В морях и океанах моллюски живут на разных глубинах. Сухопутные виды легко переносят разные климатические условия, что обусловлено их способностью впадать в спячку — зимнюю (на севере) или летнюю и зимнюю (южные регионы). При этом улитка заползает в почву, втягивается целиком в раковину и заклеивает в нее вход.

Некоторых брюхоногих моллюсков используют в пищу и они являются объектом промысла (в Черном море обитает моллюск морское блюдечко — *Patella*). Встречающаяся в нашей стране виноградная улитка

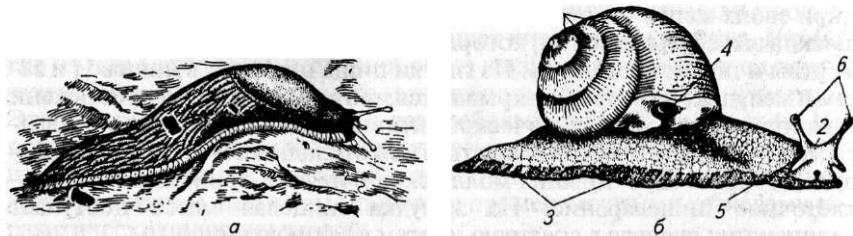


Рис. 82. Представители наземных брюхоногих моллюсков:
а — слизень оранжево-желтый; б — виноградная улитка; 1 — раковина; 2 — голова с двумя парами щупалец; 3 — нога; 4 — дыхательное отверстие; 5 — половое отверстие; 6 — глаза

(*Helix pomatia*) относится к числу вредителей виноградной лозы, а в ряде стран этих улиток употребляют в пищу, для чего специально разводят их.

Многочисленные многоядные слизни наносят вред различным растениям, поедая листья, ягоды и клубни. Тело слизней вытянуто и лишено раковины, правда, у некоторых видов под кожей имеются остатки раковины. Особый ущерб наносят полевым и кормовым культурам полевой слизень (*Agriolimax agrestis*), крупный окаймленный слизень (*Arion circumscriptus*) и другие (рис. 82). Обычно это ночные животные, днем прячущиеся в укрытиях. Слизни — гермафродиты. Дают за летний период несколько кладок по 9—50 яиц в каждой. Зимуют на стадии яйца и взрослой особи. Продолжительность жизни слизней — до 3 лет.

Наземные и пресноводные брюхоногие моллюски являются промежуточными хозяевами плоских паразитических червей — сосальщиков. Это малый прудовик (*Limnaea truncatula*), обыкновенный прудовик (*L. stagnalis*), битиния (*Bithynia leachii*), хелицелла (*Helicella candidula*) и многие другие (рис. 83). Роль этих брюхоногих моллюсков весьма вели-

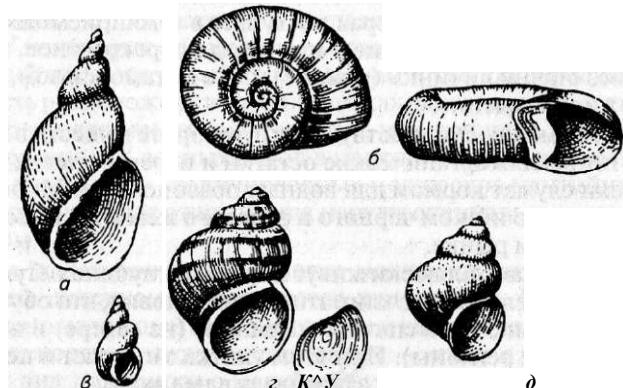


Рис. 83. Раковины различных пресноводных брюхоногих моллюсков:
а — прудовик обыкновенный (*Limnaea stagnalis*); б — катушка обыкновенная (*Planorbis corneus*); в — малый прудовик (*Limnaea truncatula*'); г — лужанка живородящая; д — битиния (*Bithynia leachii*)

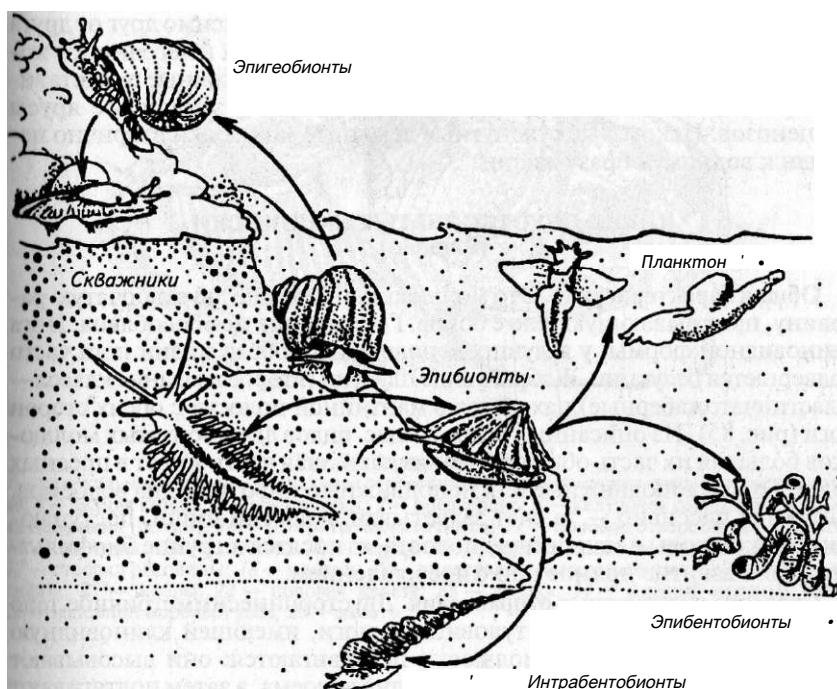


Рис. 84. Экологическая радиация брюхоногих моллюсков

ка в распространении гельминтных заболеваний, так как эти мягкотелые весьма многочисленны в природе. Например, малый прудовик живет во всех водоемах, начиная с маленьких луж и кончая крупными водоемами и достигает численности нескольких миллионов на гектар. Моллюски зимуют, зарываясь в грунт дна водоемов. При этом более половины из них могут быть поражены личинками фасциол. Моллюски служат косвенным индикатором загрязнения водоемов.

Филогения брюхоногих моллюсков. Считают, что предками брюхоногих моллюсков были небольшие примитивные гастроподы с раковиной в виде колпачка. В процессе эволюции увеличивались размеры моллюсков и раковина спирально закручивалась. Легочные и заднежаберные моллюски представляют наиболее продвинутые ветви эволюции. Экологическое разнообразие гастропод превосходит экологическое разнообразие других классов моллюсков (рис. 84).

В процессе эволюции происходило совершенствование специализации ползающих по дну, а также образование роющих видов с раковиной в виде буравчика со множеством оборотов спирали. Особый интерес представляет кораллобионтная группа моллюсков, приспособившихся к жизни на коралловых полипах. К плавающему образу жизни

перешли киленогие и крылоногие моллюски. Независимо друг от друга перешли к жизни на сушу часть гребенчатожаберных и легочных, у которых развитие стало протекать без метаморфоза. Легочные представители расселились по всем географическим зонам, заняли все ярусы биоценозов. Некоторые сухопутные легочные моллюски вторично перешли к водному образу жизни.

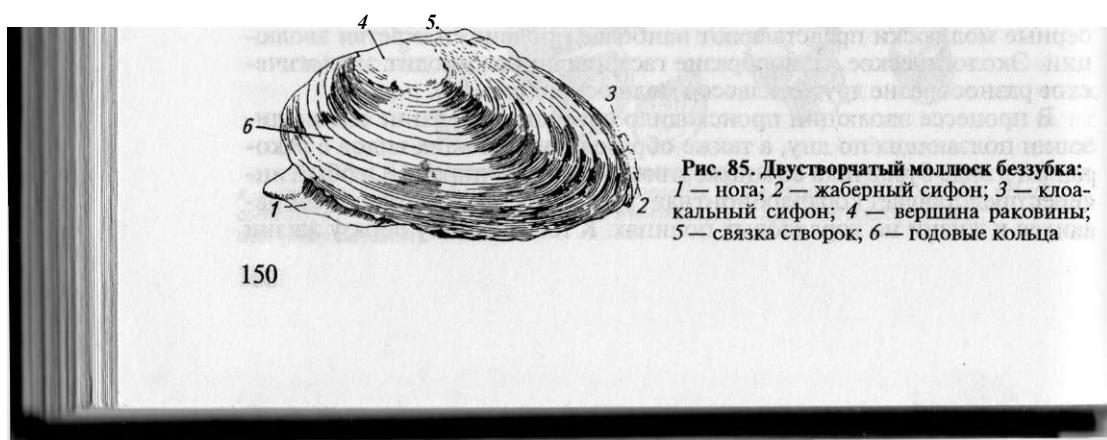
КЛАСС ДВУСТВОРЧАТЫЕ МОЛЛЮСКИ (Bivalvia)

Общая характеристика. Это моллюски, имеющие двусторчатую раковину, прикрывающую тело с боков. Голова у них обособлена. Нога клиновидной формы, у ведущих неподвижный образ жизни нога часто подвергается редукции. Жабры в виде пластин (второе название класса Пластинчатожаберные) находятся в мантийной полости с обеих сторон ноги (рис. 85). Из описанных более 20 тыс. видов двусторчатых моллюсков большая их часть обитает в морях, меньшая часть живет в пресных водах. Это малоподвижные или неподвижные животные дна водоемов. Питаются пассивно — за счет захватывания частиц пищи, поступающих с током воды в мантийную полость, относятся к группе биофильтраторов. Развитие прямое или с превращением.

Строение и жизненные отравления. Двустороннесимметричное тело этих животных состоит из туловища и ноги, имеющей клиновидную форму. С помощью ноги моллюски передвигаются: они высасывают ногу из раковины, зарывают ее в грунт дна водоема, а затем подтягивают к ней тело. У неподвижных видов нога в разной степени редуцирована.

Покровы богаты различными железами, вырабатывающими слизь, кислоту для разрушения известковых скал, материал для прикрепления к субстрату и ряд других веществ. Мантия в виде двух складок свешивается с боков тела. Между телом и мантией образуется мантийная полость, в которой расположены жабры и нога, в нее открываются задняя кишечная, мочевые и половые протоки.

В мантийную полость вода поступает через жаберный сифон, омывает жабры и удаляется через выводной (колоакальный) сифон. Движение воды обеспечивает реснички мерцательного эпителия, который покрывает мантию, жабры и сифоны. Вода приносит в мантийную полость пищевые частицы и кислород.



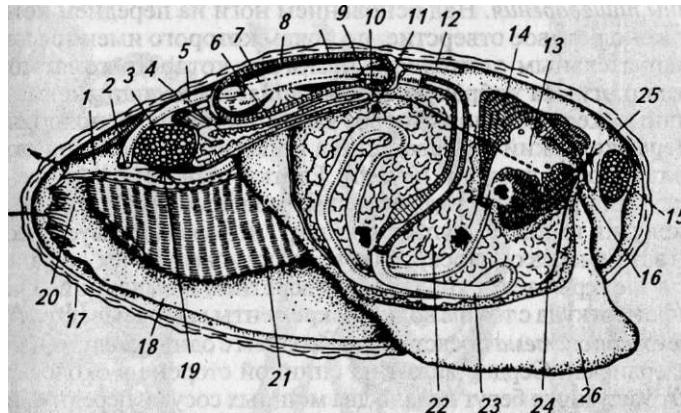


Рис. 86. Анатомия беззубки:
 / — клоакальный сифон; 2 — анальное отверстие; 3 — задний мускул-замыкатель; 4 — почки; 5 — задняя кишка; 6 — мочеточник; 7 — предсердие; 8 — желудочек сердца; Р — воронка нефридия (почки); 10 — отверстие мочеточника; 11 — половое отверстие; 12 — аорта; 13 — желудок; 14 — печень; 15 — передний мускул-замыкатель створок; 16 — рот; 17 — створка раковины; 18 — мантия; 19 — висцеральный нервный узел; 20 — жаберный сифон; 21 — жабры; 22 — половая железа; 23 — кишка; 24 — ножной нервный узел; 25 — головной нервный узел; 26 — нога

Две створки раковины соединены на вершине эластичной связкой (лигаментом) или замком, расположенными по верхнему краю створок. У большинства двустворчатых моллюсков створки имеют одинаковые размеры и форму, но у неподвижных форм они могут различаться. У живого моллюска створки раковины способны раскрываться и замыкаться. Захлопывание (закрытие) створок обеспечивают мускулы-замыкатели, связывающие обе створки. При их сокращении створки закрываются, а при расслаблении открываются за счет рогового эластичного лигамента, который действует как пружина.

Нарастание раковинных створок происходит по наружному их краю **за** счет секретов желез наружного эпителия мантии. Зимой моллюски 11 практически не растут, и поэтому на створках образуются две годичные полосы (летняя и зимняя), по которым легко определить возраст моллюска. Раковина имеет три слоя: органический, фарфоровидный (из углекислого кальция) и перламутровый.

Нервная система состоит из трех пар нервных ганглиев, расположенных над глоткой, в ноге и в задней части туловища (рис. 86) и связанных комиссарами. Органы чувств развиты слабо. В покровах разбросаны чувствующие клетки, на жабрах есть хеморецепторы, в ноге расположены органы равновесия — статоцисты. У некоторых видов по краям мантии разбросаны многочисленные глазки. Головные щупальца и глаза отсутствуют.

Органы пищеварения. Над основанием ноги на переднем конце тела расположено ротовое отверстие, по бокам которого имеются две лопасти с мерцательным эпителием, реснички которого гонят пищевые частицы ко рту. На жабрах и ротовых лопастях имеются органы вкуса и ресничные желобки, по которым частички пищи транспортируются в рот. Через короткий пищевод пища попадает в небольшой желудок, куда открываются протоки печени. Глотка, терка и слюнные железы из-за редукции головы у двустворчатых моллюсков отсутствуют.

Из желудка пища поступает в кишку, которая образует несколько петель и затем через околосердечную сумку и желудочек сердца проходит назад и открывается анальным отверстием в выводной (клоакальный) сифон, откуда с током воды экскременты выбрасываются наружу.

Кровеносная система представлена сердцем с одним желудочком и двумя предсердиями. Сердце лежит на спинной стороне в околосердечной сумке. От желудочка берут начало два мощных сосуда: передняя и задняя аорты. Из лакун полости тела венозная кровь направляется в приносящие жаберные сосуды. Окисленная артериальная кровь из жабер по выносящим сосудам возвращается в сердце. Часть крови проходит в почки, миная жабры, где освобождается от конечных продуктов обмена и вливается в выносящие жаберные сосуды, которые впадают в предсердие.

Органы выделения представлены двумя почками, которые лежат под сердцем. Каждая почка начинается выстланной мерцательным эпителием воронкой в околосердечной сумке. Мочеточники открываются в мантийную полость.

Органы размножения. Большинство двустворчатых моллюсков раздельнополы. Половые железы парные, а их протоки открываются в мантийную полость. Оплодотворение яйцеклеток наружное. Мужские половые клетки из мантийной полости самцов наружу попадают через выводной сифон. У пресноводных форм оплодотворение происходит в мантийной полости самки, куда через жаберный сифон с током воды заносятся спермии. Такое оплодотворение возможно при тесном заселении моллюсков.

Развитие у большого числа видов происходит с метаморфозом. У морских представителей личинка парусник похожа на личинку колышчатых червей: прозрачное тело округлой формы с диском, покрытым поясами ресничек, — парус.

У пресноводных моллюсков (беззубки, перловицы) личинки глохидии имеют двустворчатую раковину с зубцами по краям (см. рис. 81). Яйца откладываются в жабры, где и развиваются глохидии. Глохидии появляются осенью и всю зиму остаются в мантийной полости материнского организма. Покидают глохидии мантийную полость весной с током воды через клоакальный сифон и закрепляются на жабрах рыб. Ткани рыб обрастают глохидиев, которые превращаются в эктопаразитов, питаясь соками рыбы-хозяина. Через 2—3 мес молодые моллюски покидают хозяина, опускаются на дно и начинают самостоятельную жизнь. Эктопаразитизм глохидиев на рыбах обеспечивает расселение моллюсков в водоемах.

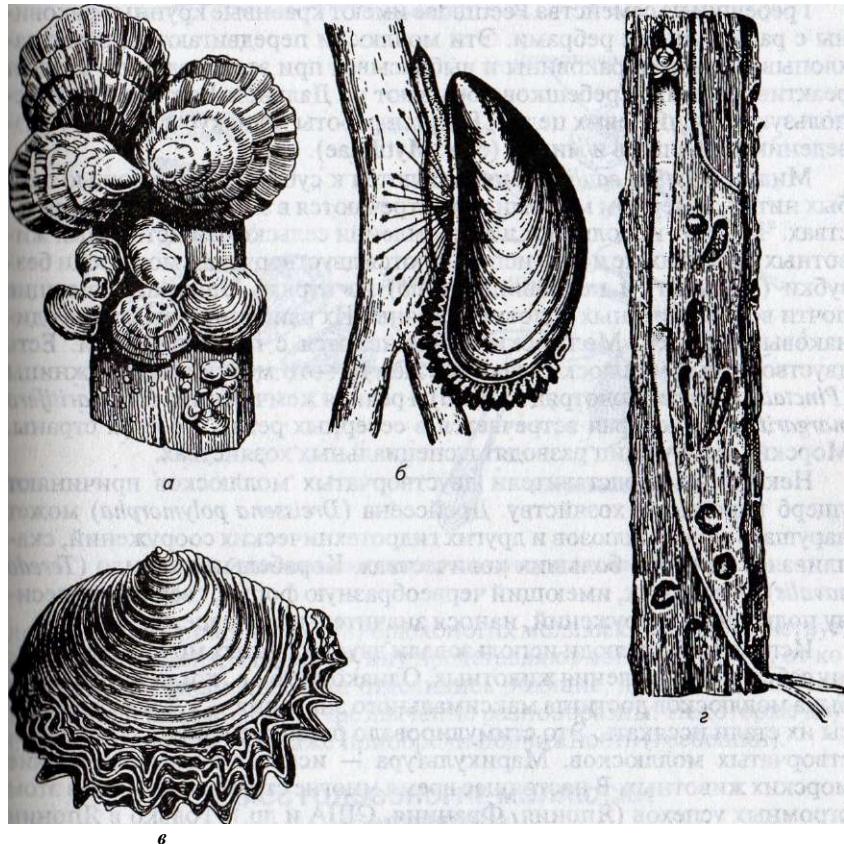


Рис. 87. Представители двустворчатых моллюсков:
а — устрицы (*Ostrea*) на куске дерева; б — мидии (*Mytilus edulis*), прикрепленная к субстрату (юмошью нитей биссуса); в — жемчужница (*Pteria*); г — корабельный червь (*Teredo navalis*)

Пластинчатожаберные моллюски являются очень эффективными естественными биофильтраторами воды. Например, при высокой плотности заселения мидии, живущие на 1 м² поверхности дна, пропускают через себя более 250 т воды.

Многих двустворчатых моллюсков используют в пищу (устрицы, гребешки, мидии и др.; рис. 87). Устрицы (сем. Ostreidae) ведут неподвижный образ жизни в морских водах, имеют асимметричную раковину. Большой выпуклой створкой они соприкасаются с субстратом, образуя огромные скопления на мелководьях. Во многих странах мясо устриц, характеризующееся высокой питательной ценностью, широко используют в пищу. В прибрежных странах эффективно развивается промышленное разведение устриц.

Гребешки из семейства Pectinidae имеют красивые крупные раковины с радиальными ребрами. Эти моллюски передвигаются, резко захлопывая створки раковины и выбрасывая при этом воду, что создает реактивную тягу. Гребешков добывают на Дальнем Востоке РФ и используют для пищевых целей. Ведутся работы по искусственноому разведению гребешков и мидий (сем. Mytilidae).

Мидии (*Mytilus edulis*) прикрепляются к субстрату с помощью особых нитей. В Черном море мидии встречаются в значительных количествах. Часто их используют для кормления сельскохозяйственных животных. Для этих же целей используются двустворчатые моллюски беззубки (*Anodonta*) и перловицы (*Unio*) из отряда Unionida, живущие почти во всех пресных водоемах страны. Их раковины имеют две одинаковые створки. Моллюски передвигаются с помощью ноги. Есть двустворчатые моллюски, дающие жемчуг: это морские жемчужницы (*Pinctada* и *Pteria*, подотряд *Pteriina*) и речная жемчужница (*Margaritifera margaritifera*), которая встречается в северных реках и озерах страны. Морских жемчужниц разводят в специальных хозяйствах.

Некоторые представители двустворчатых моллюсков причиняют ущерб народному хозяйству. Дрейссена (*Dreissena polymorpha*) может нарушать работу шлюзов и других гидротехнических сооружений, скапливаясь на них в больших количествах. Корабельное точило (*Teredo navalis*) — моллюск, имеющий червеобразную форму, сверлит древесину подводных сооружений, нанося значительные повреждения.

Испокон веков люди использовали двустворчатых моллюсков в пищу себе и для кормления животных. Однако с 1962 г., когда мировая добыча моллюсков достигла максимального значения — 1,7 млн т, ресурсы их стали иссякать. Это стимулировало развитие марикультуры двустворчатых моллюсков. Марикультура — искусственное разведение морских животных. В настоящее время многие страны добились в этом огромных успехов (Япония, Франция, США и др.). Только в Японии путем выращивания жемчужниц ежегодно получают более 100 тыс. жемчужин. Ведутся такие работы и в нашей стране.

Двустворчатые моллюски выполняют и очистительные функции в водоемах. Они поглощают и накашивают в своем теле тяжелые металлы и очищают воду от химических загрязнений. В среднем один моллюск пропускает за 1 ч около 1 л воды. Перловицы и беззубки являются действующими биофилtrаторами. В этом отношении они представляют большую ценность, чем как источник мяса.

Филогения двустворчатых моллюсков. Наиболее примитивны по своей организации первичножаберные двустворчатые моллюски, которые и берут начало от древних первичножаберных предков. В дальнейшем в процессе эволюции при переходе к неподвижному образу жизни у моллюсков редуцировалась нога; у перегородчатожаберных редуцировались жабры, а функции дыхания перешли к наджаберным полостям.

У предков двустворчатых раковина, видимо, была цельной. Жизнь на субстрате заставила моллюсков защитить свое тело по бокам. Эко-

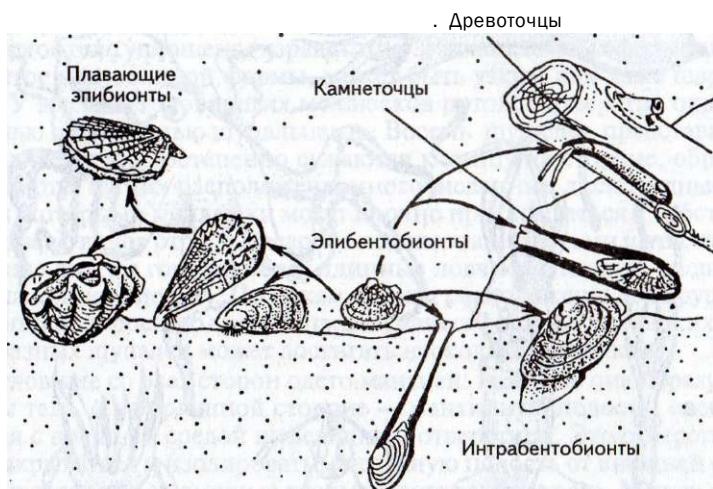


Рис. 88. Экологическая радиация двустворчатых моллюсков

логическая радиация (рис. 88) брюхоногих моллюсков свидетельствует о том, что их центральную группу представляют донные формы, от которых путем специализации отделились роющие, древо- и камнеточцы. Неподвижные формы чрезвычайно разнообразны. Некоторые двустворчатые моллюски даже приобрели подвижность (гребешки).

КЛАСС ГОЛОВОНОГИЕ МОЛЛЮСКИ (Cephalopoda)

Общая характеристика. К этому классу принадлежит около 700 видов крупных моллюсков, живущих исключительно в морях и отличающихся наиболее сложной организацией. Из-за совершенных приспособлений к жизни в морской стихии и сложности поведения головоногих моллюсков часто называют «приматами моря» среди беспозвоночных животных. Обычно это свободноплавающие и подвижные хищники, предпочитающие воды теплых морей и океанов. Среди них мало ползающих видов. Их размеры колеблются от нескольких сантиметров до 18 м (гигантские кальмары).

Тело отчетливо подразделяется на голову и туловище. Нога же превращена в щупальца (руки), которые вторично сместились на голову и окружают ротовое отверстие (отсюда и их название — головоногие). Другая часть ноги преобразовалась в воронку, лежащую у входа в мантийную полость на брюшной стороне тела.

У примитивных форм раковина наружная, у высших представителей она внутренняя, может быть частично или полностью редуцирована.

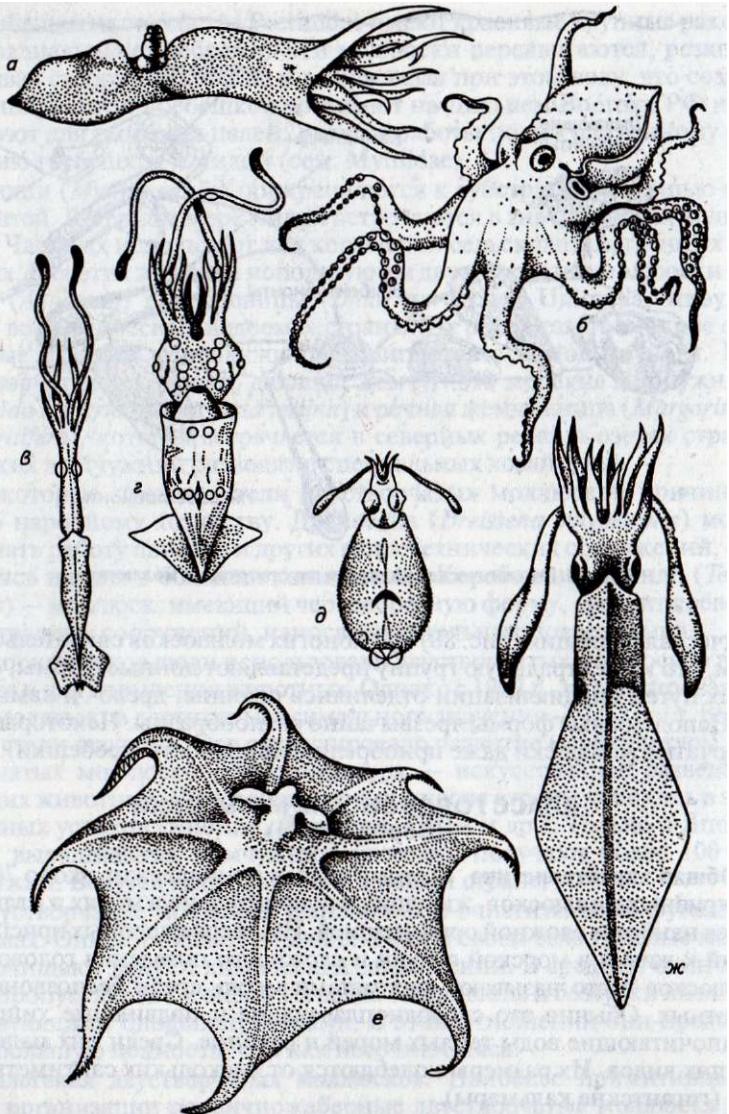


Рис. 89. Различные головоногие:
 а — глубоководный плавающий осьминог (*Amphitretus pelagicus*); б — осьминог (*Benthoctopus profundorum*); в — планктонный кальмар (*Doratopsis sagitta*); г — глубоководный пелагический кальмар со светящимися органами (*Lycoteuthis diadema*); д — планктонный кальмарчик (*Cranchia scabra*); е — донный осьминог (*Cirrothauma murrayi*); ж — пелагический кальмар (*Loligo edulis*)

Строение и жизненные отправления. Моллюски, обитающие в толще воды, имеют тело торпедовидной формы (кальмары), у бентосных форм тело мешкообразной формы (осьминоги), у других универсальных видов тело уплощено (каракатицы). У планктонных форм тело студенистое медузиодной формы, может быть узким или даже шаровидным. У высших головоногих моллюсков ротовое отверстие окружено восемью или десятью щупальцами. Восемь щупалец представителей отряда Octopoda постепенно сужаются к концу на стороне, обращенной ко рту, на них расположены многочисленные дисковидные присоски, которыми моллюски могут прочно присасываться к субстрату и к жертве. У видов отряда Decapoda помимо таких восьми щупалец имеется еще два, но гораздо более длинные ловчие щупальца, расширенные на конце (рис. 89). По бокам головы расположены два крупных и сложных по строению глаза. У примитивных форм число гладких и червеобразных щупалец может достигать нескольких десятков.

Туловище со всех сторон одето мантией: на спине она образует покровы тела, а на брюшной стороне — мантийную полость, сообщающуюся с внешней средой щелевидным отверстием. Это отверстие может закрываться и изолировать мантийную полость от внешней среды. Закрывается оно с помощью особых «застежек-кнопок». Между «кнопками» на брюшной стороне из этой щели выступает воронка в виде мускулистой трубы. Расширенный конец воронки открывается в мантийную полость, а узкий — наружу. Воронка (производное ноги) служит для особого реактивного движения. Когда мантийная щель закрыта замыкателями с помощью многочисленных мышц, мантия прижимается к туловищу. Вода из мантийной полости с силой выталкивается через воронку, толкая моллюска в обратную сторону (реактивная тяга). Воронка может изгибаться в разные стороны, что позволяет моллюску менять направление движения. Роль дополнительного руля выполняют щупальца и плавники в виде складки кожи. Ритмические сокращения мантии и выталкивания воды позволяют моллюску не только плавать, но и интенсивно омывать жабры водой.

В мантийную полость на брюшной стороне головоногих моллюсков открываются половые и мочевыводящие протоки, а также анальное отверстие (рис. 90).

У современных головоногих раковина сильно редуцирована и обрастает боковыми складками мантии, становясь внутренней. У некоторых представителей (каракатица *Sepia*) раковина в виде известковой пластинки залегает под покровами на спинной стороне туловища. У кальмара (*Loligo*) от раковины остается лишь скрытый под покровами спинной роговой листок. У некоторых видов раковина остается лишь у самок или исчезает вовсе.

Покровы представлены однослойным эпителием и слоем соединительной ткани под ним. Головоногие моллюски способны к быстрой и резкой смене своей окраски, что обуславливается наличием в соединительнотканном слое кожи многочисленных пигментных клеток — хроматофоров. Механизм смены окраски контролируется нервной сис-

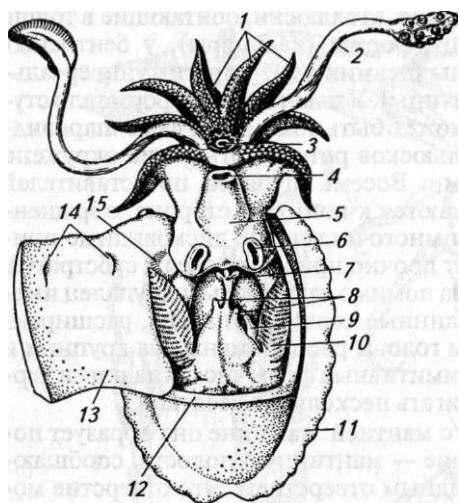


Рис. 90. Каракатица *Sepia officinalis* со вскрытой мантийной полостью; вид с брюшной стороны:

1 — руки с присосками; 2 — ловчая рука; 3 — рот; 4 — отверстие воронки; 5 — воронка; 6 — хрящевые ямки кнопок; 7 — анальное отверстие; 8 — почечные сосочки; 9 — непарный половой сосочек; 10 — жабры; 11 — плавник; 12 — линия отреза мантии; 13 — отогнутая мантия; 14 — хрящевые бугорки кнопок; 15 — мантийный звездчатый ганглий

темой, получающей информацию по зрительным нервам.

Нервная система головоногих моллюсков устроена наименее сложно. Нервные ганглии образуют крупное окологлоточное скопление — мозг,

заключенный в хрящевую капсулу (соответствует по выполняемой функции черепу позвоночных). От заднего отдела ганглиозной массы отходят два крупных мантийных нерва.

Органы чувств хорошо развиты: обонятельные ямки под глазами, обладающие высокой чувствительностью, пара статоцистов внутри хрящевой головной капсулы, крупные и сложно устроенные глаза, способные к аккомодации. Глаза по своему строению напоминают глаза млекопитающих (пример конвергенции между беспозвоночными и позвоночными животными). Глазное яблоко сверху покрыто роговицей, имеющей отверстие в переднюю камеру глаза. Радужная оболочка образует отверстие — зрачок, через который свет попадает на хрусталик. Аккомодация глаза происходит за счет удаления хрусталика от сетчатки или приближения его (у млекопитающих аккомодация осуществляется путем изменения кривизны хрусталика). Глаза окружены хрящевой капсулой. На коже имеются особые органы свечения, по строению напоминающие глаза.

Органы пищеварения также сложно устроены и несут черты специализации к питанию животной пищей. Ротовое отверстие, лежащее в центре венца щупалец, ведет в мускулистую глотку, в которой находится язык с теркой. В глотке расположены две толстые роговые челюсти, загнутые в виде крючка и напоминающие клюв попугая. В глотку открываются протоки одной-двух пар слюнных желез, секрет которых обладает амилолитической и протеолитической активностью, может содержать яды. Головоногие моллюски питаются только полужидкой пищей, поскольку у них узкий пищевод, который проходит через мозг моллюска. Пища сначала разгрызается роговыми челюстями, а затем обильно смачивается слюной и перетирается теркой. Длинный пищевод может иметь расширение — зоб.

Из пищевода пища попадает в мускулистый энтодермальный желудок, имеющий слепой мешковидный отросток. От желудка отходит тонкая кишка, переходящая в заднюю кишку, оканчивающуюся анальным отверстием в мантийную полость. В желудок впадают протоки печени, секрет которой имеет весь набор пищеварительных ферментов. Гистия и поджелудочная железа в виде небольших придатков в протоках печени.

В заднюю кишку перед анальным отверстием открывается проток чернильного мешка, в котором образуется черная жидкость. Выбрасывая эту чернильную жидкость через анальное отверстие, а затем и из мантийной полости через воронку наружу, моллюски окружают себя темным облаком, что позволяет им скрыться от врагов. Питаются головоногие моллюски в основном рыбой, крабами и двустворчатыми моллюсками, схватывая их щупальцами и убивая челюстями и ядом.

Органы дыхания — жабры, расположенные в мантийной полости симметрично по бокам туловища. Обмен воды осуществляется сокращением мантийных мышц и работой воронки, через которую вода выталкивается наружу. По числу жабр головоногие моллюски делятся на две группы: четырехжаберные (*Tetrabranchia*) идвужаберные (*Dibranchia*).

Кровеносная система представлена сердцем с одним желудочком и двумя или четырьмя предсердиями (по числу жабр). Кровь движется за счет сокращений сердца, а также за счет пульсации участков сосудов. От переднего и заднего концов желудочка сердца отходят головная и мнутренностная аорты. Капилляры вен и артерий в коже и мышцах переходят друг в друга и лишь в некоторых местах сохраняются лакунарные пространства; таким образом, кровеносная система почти замкнутая. Кровь на воздухе голубеет, поскольку содержит гемоцианин (богатое мембрено соединение, соответствующее по физиологическим функциям гемоглобину позвоночных).

Выделительная система состоит из двух или четырех почек, берущих начало отверстиями в целоме (околосердечной сумке). Конечные продукты обмена поступают из жаберных вен и околосердечной сумки и выделяются в мантийную полость рядом с анальным отверстием.

Половая система. Головоногие моллюски — раздельнополые животные, у которых часто хорошо выражен половой диморфизм. Половые железы и их протоки непарные. Половые продукты накапливаются в целоме и выводятся через половые протоки. Спермин склеиваются в тинерматофоры — пакеты с плотной оболочкой.

Оплодотворение обычно происходит в мантийной полости самки, роль копулятивного органа играет одно из щупалец, которое у самцов отличается наличием особого ложкообразного придатка. С помощью этого щупальца самец вводит сперматофоры в мантийную полость самки. Все развитие зародышей протекает внутри яиц, которые самка откладывает на дне. У некоторых головоногих проявляется забота о потомстве: самка аргонавта вынашивает яйца в выводковой камере, осьминоги охраняют кладку яиц.

Современные головоногие относятся к двум подклассам: подкласс Наутилиды (*Nautiloidea*) и подкласс Колеоиды (*Coleoidea*).

Головоногие моллюски отличаются крупными размерами: от нескольких сантиметров до нескольких метров. Удалось обнаружить 10-метровое щупальце головоногого моллюска. Живут моллюски только в морях и ведут разнообразный образ жизни. Большинство относятся к пелагическим животным, живущим в толще воды. У донных видов (часть осьминогов) между щупальцами есть перепонка, придающая телу моллюска вид диска, лежащего на дне. Все головоногие — хищники, нападающие на ракообразных и рыб, которых они схватывают щупальцами, убиваючи ими челюстями и ядом слюнных желез.

Многие головоногие являются объектом промысла: кальмаров, каракатиц и осьминогов человек использует в пищу, поскольку их мясо обладает высокой пищевой ценностью. Мировой улов головоногих достигает более 1,6 млрд т в год.

Наутилиды включают лишь один отряд *Nautilida*, к которому относится всего несколько видов, обитающих в тропических областях океанов. Наутилиды характеризуются многими примитивными чертами: наружная многокамерная раковина, многочисленные без присосок щупальца, проявление метамерии и т. п. Наутилус плавает реактивным способом. Является объектом промысла из-за красивой раковины.

Подкласс Колеоиды (*Coleoidea*) включает около 650 видов жестко-кожих моллюсков, лишенных раковины. У них сросшаяся воронка и щупальца вооружены присосками, кроме того, у них две жабры, две почки и два предсердия.

Характерным представителем отряда являются каракатицы (*Sepia*), имеющие десять щупалец, из которых два ловчие. Обитают вблизи дна и ведут активный плавающий образ жизни.

К отряду Кальмары (*Teuthida*) относятся многие промысловые виды (*Todarodes*, *Loligo* и др.) У них иногда сохраняетсяrudimentарная раковина в виде роговой пластинки под кожей. Кальмары имеют десять щупалец. Это торпедовидные обитатели толщи океанских вод.

Следов раковины нет у наиболее эволюционно прогрессивных головоногих моллюсков — представителей отряда Восьминогие (*Octopoda*). У них восемь щупалец, одно из которых у самцов превращено в половое. Большинство осьминогов обитают в придонном слое воды. Среди осьминогов есть представители, имеющие выводковую камеру (аргонавт).

Филогения головоногих моллюсков. Самые древние представители головоногих — наутилиды, раковины которых обнаруживаются в ископаемых кембрийских отложениях. Считается, что головоногие произошли от древних ползающих раковинных моллюсков. В процессе эволюции сформировалась группа головоногих моллюсков, лишенных раковины, с новым реактивным типом движения, со сложной нервной системой и сложными органами чувств.

От примитивных раковинных бенто-пелагических форм определилось несколько путей экологической специализации (рис. 91). Проис-

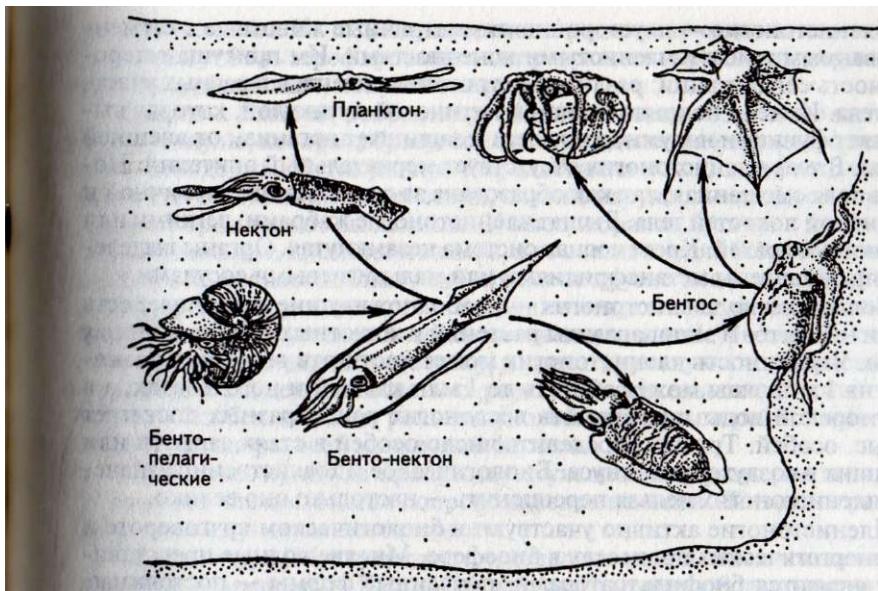


Рис. 91. Экологическая радиация головоногих моллюсков

ходит переход к бенто-нектонным формам, у которых раковина становится внутренней и ее функция как плавательного аппарата ослабевает, но развивается новая модель движителя — воронка. Они-то и дали начало безраковинным моллюскам, которые образуют бенто-нектонные (каракатицы, осьминоги), нектонные (кальмары, осьминоги и каракатицы), бентосные и планктонные (зонтикообразные осьминоги, палочковидные кальмары) формы.

ТИП ЧЛЕНИСТОНОГИЕ (Arthropoda)

Общая характеристика. Это наиболее распространенная и процветающая группа животных, обладающих членистыми конечностями и сегментированным телом. Насчитывают более 1,5 млн видов членистоногих, населяющих моря, океаны, пресные водоемы, поверхность суши, почву и воздушную среду. Среди них много паразитов. Членистоногие освоили все способы движения, среди них имеются плавающие, роющие, ползающие, бегающие, летающие, прыгающие, реже малоподвижные и неподвижные формы. Чрезвычайно разнообразен спектр их питания: от обычной животной или растительной пищи до самых труднопереваримых объектов — древесины, рога, волоса, пера и т. п.

Членистоногие — двусторонне-симметричные животные с сегментированным телом и членистыми конечностями. Им присуща гетерономность сегментации: различное строение сегментов в разных участках тела. Их тело покрыто прочной хитиновой кутикулой, которая выполняет функции наружного скелета и защищает организм от внешней среды. В теле членистоногих отсутствует мерцательный эпителий. Половость тела смешанная, так как образована за счет слияния первичной и вторичной полостей тела. Дышат членистоногие жабрами, легкими или с помощью трахей. Кровеносная система незамкнутая. Органы выделения представлены метанефридиями или мальпигиевыми сосудами.

Большинство членистоногих — свободноживущие животные, есть среди них экто- и эндопаразиты растений и животных. Много хищных форм. Численность членистоногих может достигать огромных размеров: на 1 м² почвы может обитать до 1 млн клещей и ногохвосток, а в 1 м³ морской воды численность веслоногих ракообразных достигает 30 тыс. особей. Трудно определить число особей в стаях саранчи или парящих в воздухе тучах гусениц. Биологическое и хозяйственное значение членистоногих нельзя переоценить — настолько оно велико.

Членистоногие активно участвуют в биологическом круговороте и биоэнергетических процессах в биосфере. Многие водные представители являются биофильтраторами, почвенные формы — постоянные участники почвообразовательных процессов. Среди членистоногих много объектов промысла и промышленного разведения, постоянно вовлекаются в сельскохозяйственное производство новые виды насекомых (пчелы, шмели, наездники, осы, хищные формы, паразитические виды и др.). Среди членистоногих есть опасные вредители лесов и сельскохозяйственных культур, паразиты и переносчики заболеваний животных и человека.

Тип Членистоногие подразделяют на четыре подтипа, из которых наибольший интерес представляют три подтипа: Жабродышащие (Branchiata), Хелициеровые (Chelicera) и Трахейные (Tracheata).

Строение и жизненные отправления. Форма тела членистоногих чрезвычайно разнообразна. Тело состоит из сегментов, следующих друг за другом. Каждый сегмент покрыт четырьмя склеритами (твердыми хитиновыми пластинками): спинная пластинка — тергит, брюшная пластинка — стернит и две боковые пластинки. Между склеритами расположены мягкие сочлененные мембранны, придающие подвижность каждому сегменту. У некоторых представителей сегменты сходны, но у большинства видов они различны, что позволяет выделить отделы тела: голову, грудь и брюшко. Головной отдел состоит из акрона и четырех сегментов. У некоторых представителей сегменты тела сливаются в нерасчлененные головогрудь и брюшко, а у клещей сегментация тела не выражена, и они имеют нерасчлененное тело.

Как правило, каждый сегмент тела членистоногих несет по паре членистых конечностей, что и определило название типа. Конечности подвижно соединяются с телом с помощью суставов и состоят из нескольких члеников, образуя многоколенный рычаг, способный к силь-

ным движениям. На отдельных сегментах или даже отделах тела конечности могут быть атрофированы или превращены в различные органы — **ротовые**, яйцеклады, копулятивные и др.

Покровы. Главная особенность членистоногих — наличие хитиновой кутикулы, которая образуется за счет выделения наружного покрова — гиподермы. Кутикула прочна, эластична и защищает тело членистоногих от внешних воздействий, служит наружным скелетом, предохраняет организм от высыхания. У высших раков кутикула пропитана солями кальция, кремния и железа, в ее состав входят белковые, жиро-подобные, воскоподобные, дубильные и другие вещества. Особенно характерен для кутикулы полисахарид хитин, в молекуле которого в отличие от других углеводов содержатся атомы азота. По своему строению хитин напоминает клетчатку, инкрустирующую клетки растений, что может свидетельствовать о родстве растительного и животного царств. Особенno важна роль хитиновой кутикулы в работе конечностей и как опоры всей мышечной системы.

Хитиновая кутикула препятствует увеличению размеров тела, поэтому рост членистоногих сопровождается линькой. Животное растет, пока новая кутикула не затвердеет.

Мускулатура. Мускулатура представлена отдельными мышечными пучками — мышцами. Кожно-мускульного мешка у большинства членистоногих нет. Мыши прикреплены к наружному покрову. Мускулатура внутренних органов представлена гладкими мышечными волокнами, а остальная мускулатура имеет поперечно-полосатую структуру, что отличает членистоногих от червей, имеющих гладкую мускулатуру. Строение мускулатуры членистоногих является примером конвергенции с позвоночными животными. Но относительная сила мышц членистоногих значительно выше, чем у млекопитающих. Мышцам членистоногих свойственна весьма высокая частота сокращений (крыловые мышцы насекомых).

Полость тела — смешанная (миксоцель). Появление членистых конечностей и поперечно-полосатой мускулатуры позволило многим членистоногим вести активный образ жизни. Каждый членик ноги, **любой** сегмент приводится в движение определенными мышцами, работу которых контролирует нервная система.

Нервная система во многом идентична нервной системе кольчатых Червей. Над глоткой в головном отделе расположены парные надглоточные ганглии, образующие хорошо развитый мозг. Имеются около-Глоточное нервное кольцо и брюшная нервная цепочка. При слиянии ряда сегментов тела соответственно происходит слияние и нервных узлов брюшной цепочки. Узлы брюшной нервной цепочки членистоногих выполняют разные функции, так как у этих животных происходит Дифференцировка тела на разные отделы. Этим членистоногие отличаются от кольчатых червей, у которых функции всех узлов нервной цепочки одинаковы.

Для членистоногих характерно сложные поведение и ориентация В пространстве, что привело к развитию высокоспециализированных

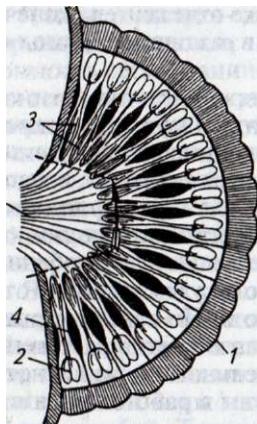


Рис. 92. Схема строения сложного глаза членистоногого:
1 — роговица; 2 — хрустальный конус; 3 — светочувствительные элементы глаза;
4 — пигментированные прослойки между омматидиями; 5 — зрительный нерв

ется на лежащие ниже светочувствительные клетки. Омматидии отделены друг от друга пигментными клетками, которые позволяют различать не только форму, но и цвета предметов. Каждый омматидий воспринимает лишь часть предмета, так как их нервные клетки воспринимают только те лучи, которые падают перпендикулярно к поверхности омматидия. Поэтому изображение предмета в сложном глазу составляется из отдельных частей (как мозаика). Отсюда и название такого зрения — мозаичное. У некоторых раков сложные глаза расположены на особых стебельках.

Хорошо развиты у членистоногих органы равновесия, слуха, осязания и обоняния. У высших членистоногих обоняние достигает совершенства.

Органы пищеварения. Ротовое отверстие находится на головном отделе и вооружено видоизмененными конечностями — ротовыми частями, позволяющими добывать, удерживать, размельчать, переваривать и заглатывать пищу. Строение и форма ротовых частей чрезвычайно разнообразны. Пищеварительная система имеет три отдела. В переднем отделе обособлены глотка и пищевод, а также жевательный желудок, покрытые кутикулой. В среднем отделе происходят процессы переваривания и всасывания пищи. Роль пищеварительных желез выполняют печень или особые пилорические придатки. Задний отдел достигает значительной длины и заканчивается анальным отверстием. Передний и задний отделы имеют эктодермальное происхождение и выстланы изнутри хитиновой кутикулой, предохраняющей внутреннюю поверхность кишечника от травмирования жесткими остатками пищи.

органов чувств. У многих членистоногих наблюдается сложное инстинктивное поведение, а высшие членистоногие способны к быстрому образованию новых условных рефлексов. Особенно сложные формы поведения и взаимоотношения присущи общественным насекомым.

У большинства представителей членистоногих имеются простые или сложные глаза или одновременно и те, и другие. *Простые глаза* имеют форму бокала, устье которого закрыто хрусталиком. Дно бокала выстлано светочувствительными клетками, от которых отходит зрительный нерв. Свет через хрусталик направляется на светочувствительный слой клеток.

Сложные глаза состоят из множества глазков — омматидиев, имеющих коническую или цилиндрическую форму (рис. 92). Наружная часть омматидиев представлена прозрачной роговицей (прозрачная часть хитиновой кутикулы). Под роговицей расположен конусовидный прозрачный хрусталик, через который луч света направляется на светочувствительные клетки. Омматидии отделены друг от друга пигментными клетками, которые позволяют различать не только форму, но и цвета предметов. Каждый омматидий воспринимает лишь часть предмета, так как его нервные клетки воспринимают только те лучи, которые падают перпендикулярно к поверхности омматидия. Поэтому изображение предмета в сложном глазу составляется из отдельных частей (как мозаика). Отсюда и название такого зрения — мозаичное. У некоторых раков сложные глаза расположены на особых стебельках.

Органы дыхания. У основной массы водных обитателей — жабры, роль наружных покровов в газообмене, как правило, незначительна. Снижение роли наружных покровов в дыхании объясняется наличием плотной хитиновой кутикулы, через которую проникновение кислорода ограничено или просто невозможно, если кутикула уплотнена и пропитана минеральными солями. Даже при достаточной проницаемости кутикулы потребность в кислороде у ведущих активный образ жизни членистоногих в большинстве случаев не может быть удовлетворена только за счет дыхания через покровы тела. Это возможно лишь у небольших по размерам членистоногих, у которых очень тонкая кутикула и относительно большая площадь поверхности тела.

У сухопутных и некоторых водных членистоногих органами дыхания служат легкие или трахеи. Легкие представляют собой тонкостенные мешки, внутри которых имеются многочисленные тонкие листочки. Через покровы этих листочек и происходит газообмен. Трахеи имеют вид ветвящихся трубочек, которые открываются во внешнюю среду специальными отверстиями — дыхальцами (рис. 93). Спадаться при дыхании им не позволяют упругие спиральные хитиновые нити. Конечные разветвления трахей лишены этих нитей и проникают во все ткани и органы членистоногих, снабжая их кислородом воздуха. Только у части членистоногих дыхание осуществляется через покровы тела.

Кровеносная система у членистоногих незамкнутая, что обусловлено наличием смешанной полости тела. Активизация жизнедеятельности требует ускорения транспорта питательных веществ, что способствует

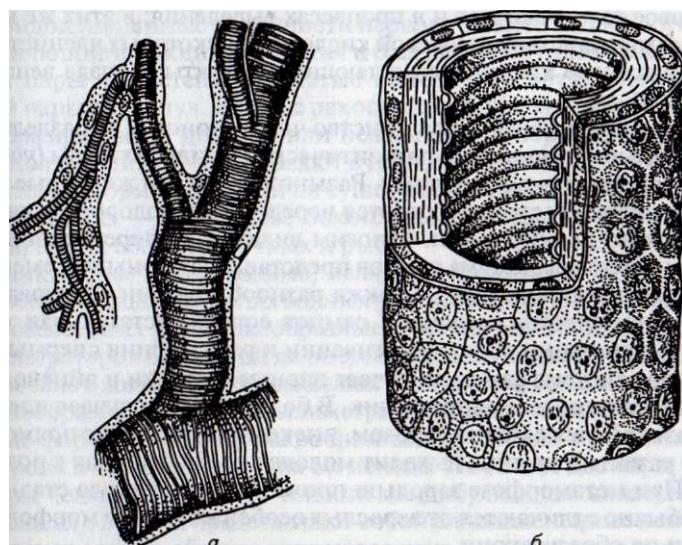


Рис. 93. Трахеи насекомых:
а — участок трахеи; б — микроскопическое строение трахеи

появлению специального органа — сердца, развивающегося из части спинного сосуда. Сердце расположено на спинной стороне животного. Сердце сокращается очень часто — до ста сокращений в минуту, что обеспечивает быстрый оборот крови (гемолимфы). Из сердца гемолимфа выталкивается через артерии в синусы миксоцеля, где омывает все внутренние органы. Обратно к сердцу гемолимфа проходит по лакунам, откуда засасывается через специальные отверстия — остии — с закрывающимися клапанами, которые имеются в стенках сердца. Гемолимфа — жидкость, частично соответствующая крови и частично целомической жидкости, выполняющая функции крови.

У некоторых членистоногих кровеносная система отсутствует или представлена одним сердцем. У трахейнодышащих функции крови ограничиваются доставкой всосавшихся в кишечнике питательных веществ. Цвет крови чаще всего желтоватый, но может быть красным, голубым и др. В крови содержатся различные клетки.

Органы выделения. У ракообразных выделительные функции выполняют видоизмененные метанефриды — почки, расположенные в головном отделе. У насекомых и многоножек органами выделения являются мальпигиевые сосуды — тонкие слепые отростки кишечника. Через стенки мальпигиевых сосудов происходит удаление конечных продуктов обмена и излишков воды. У паукообразных имеются и метанефриды, и мальпигиевые сосуды.

У многих членистоногих существенную роль в обмене веществ играет жировое тело. Оно представляет собой скопление крупных клеток, в пазме которых образуются капли жира как энергетического резерва. Но жировое тело участвует и в процессах выделения: в этих же клетках скапливаются кристаллы мочевой кислоты. У некоторых членистоногих есть специальные клетки, поглощающие продукты распада веществ — нефроциты.

Органы размножения. Большинство членистоногих — раздельнополые животные, и лишь часть паразитических и сидячих форм (усоногие раки) являются гермафродитами. Размножаются эти животные только половым путем. Часто наблюдается чередование полового и партеногенетического размножения. Многим видам характерен половой диморфизм. Половая система самцов представлена парными семенниками и двумя семяпроводами, а также разнообразными образованиями для передачи спермы самкам. У самцов есть предстательная железа, секретирующая жидкость для активации и разжижения спермы.

Половая система самок включает парные яичники и яйцеводы, непарное влагалище и семяприемник. В большинстве случаев членистоногие развиваются с метаморфозом, у некоторых развитие прямое. При прямом развитии из яйца выходит молодая особь, сходная с родительскими. При метаморфозе зародыш проходит личиночную стадию; личинки обычно отличаются от взрослых особей не только морфологически, но и по образу жизни.

Рост и развитие членистоногих связаны с периодическими линьками. Хитиновая кутикула после ее образования быстро затвердевает.

Поэтому увеличение размеров тела может происходить только после сбрасывания старой кутикулы и до затвердевания новой. Иными словами, рост членистоногих происходит периодически и тесно связан с периодичностью линьки. Перед началом линьки между кутикулой и телом членистоногих образуется жидкость, которая растворяет часть старой кутикулы (эндокутикула). Часть нерастворившейся кутикулы (эктокутикула) лопается и сбрасывается животным. Членистоногие в период линьки становятся беззащитными и поэтому почти не питаются и прячутся в укромных местах.

ПОДТИП ЖАБРОДЫШАЩИЕ (*Branchiata*)

Из обширного подтипа жабродышащих водных членистоногих в настоящее время сохранился только один класс — Ракообразные, насчитывающий более 40 тыс. видов.

КЛАСС РАКООБРАЗНЫЕ (*Crustacea*)

Общая характеристика. Основная масса ракообразных — обитатели соленых и пресных водоемов, и лишь немногие живут во влажных местах на суше (мокрицы). Тело ракообразных делится на голову, грудь и брюшко. Часто голова и грудь, сливаясь, образуют головогрудь. На голове имеются две пары усиков: антеннулы — придатки акрона, и антены — видоизмененные конечности первого сегмента головного отдела, выполняющие функции обоняния и осознания. На голове располагаются три пары челюстей. Членистые конечности двуветвистые, кроме первой пары антеннул. Водные ракообразные дышат жабрами. Многие ракообразные ведут донный или пелagicкий образ жизни. Они активно ползают по дну или плавают в толще воды, но встречаются и прикрепленные формы. К жизни на суше хорошо приспособились мокрицы, в тропиках — почвенные бокоплавы и наземные формы крабов. **Есть паразиты беспозвоночных и рыб.**

Планктонные ракообразные, являясь фитофагами, служат важнейшим звеном в пищевых цепях водных экосистем, составляя основу пищи для промысловых рыб. Ракообразные — самая многочисленная группа биофильтраторов воды. Они являются важным объектом промысла.

Строение и жизненные отравления. Размеры и форма тела чрезвычайно разнообразны: от долей миллиметра (обитатели толщи воды) до метра (донные формы). Голова у ракообразных образована в результате слияния акрона и четырех передних сегментов. На голове две пары усиков и три пары челюстей (верхние челюсти мандибулы и две нижние челюсти — максиллы), все они представляют собой видоизмененные конечности.

Сегменты груди обычно сливаются друг с другом или с головой, образуя головогрудь. У высших раков голову и грудь сверху и по бокам закрывает хитиновый щит — карапакс, защищающий жабры (рис. 94, 95).

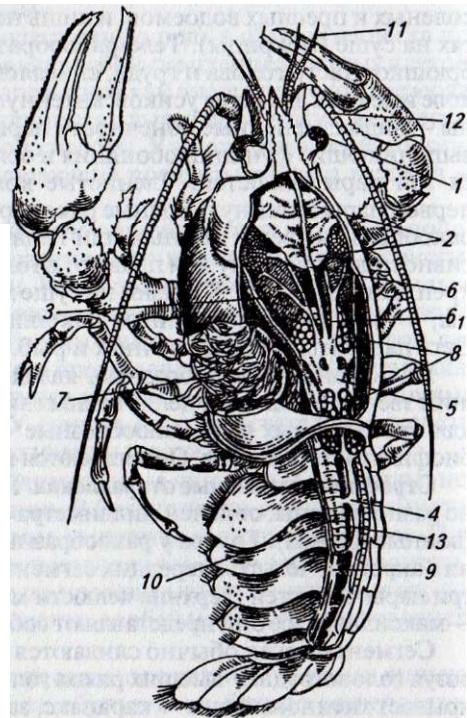
Рис. 96. Конечности самца речного рака:

1 — первая пара усиков; 2 — вторая пара усиков;
3 — жвалы (верхние челюсти); 4,5 — первая и вторая па-
ры нижних челюстей; 6—8 — ногочелюсти; 9—13 — хо-
дильные ноги; 14—19 — брюшные конечности



Рис. 97. Вскрытая самка речного рака:

1 — сложный глаз; 2 — желудок; 3 — печень;
4, 6, 6₁ — кровеносные сосуды;
5 — сердце; 7 — жабры; 8 — яичник;
9 — брюшная нервная цепочка; 10 — мышцы брюшка;
11 — первая пара усиков; 12 — вторая пара усиков; 13 — задняя кишка



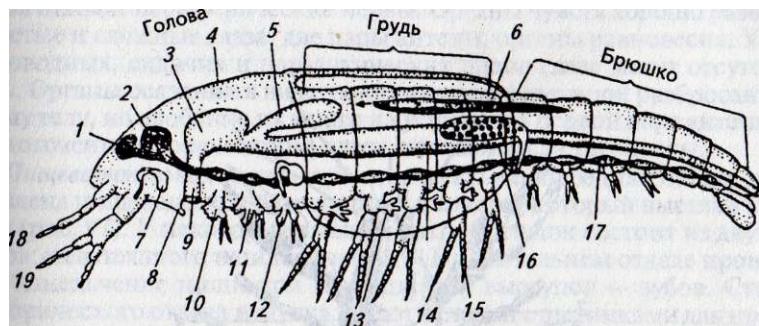


Рис. 95. Схема организации ракообразных:
1, 2—мозг; 3—желудок; 4—голова; 5—печень; 6—сердце; 7— почка; 8—губа; 9—рот;
10—мандибулы; 11, 12—максиллы; 13—эпиподит; 14, 15—экзоподит и эндоподит;
16—гонада; 17—брюшная нервная цепочка; 18—антенула; 19—антенна

Каждый сегмент груди несет по паре членистых конечностей, различающихся по форме и строению в зависимости от их функций.

Сегменты брюшка обычно не сливаются, и у высших раков каждый сегмент несет по паре ножек. У низших раков на брюшке ножек нет. Брюшко заканчивается анальной лопастью — тельсоном. У крабов брюшной отдел редуцирован. Конечности ракообразных выполняют самые разные функции (рис. 96). Они служат опорой при хождении, используются для плавания, захвата и измельчения пищи, защиты, при спаривании и т. п. Ноги имеют основную непарную часть (протоподит) и две ветви (наружная — экзоподит и внутренняя — эндоподит). Такая двуветвистая форма конечностей ракообразных сходна с двулопастной формой параподий многощетинковых кольчатых червей, но ноги ракообразных состоят из ряда члеников, что обеспечивает высокую подвижность этих животных.

Двуветвистые конечности, покрытые щетинками, характеризуются большой поверхностью и поэтому удобны для использования в качестве весел. У крупных раков ветви задней пары ног превратились в две широкие пластинки, которые вместе с широким последним члеником брюшка хорошо действуют при загребании воды брюшком.

Покровы. Наружным скелетом служит хитиновая кутикула, которая у высших раков пропитывается карбонатом кальция и превращается в прочный панцирь. У низших раков кутикула тонкая и прозрачная. Кутикула состоит из двух слоев: внутреннего — эндокутикулы и наружного — эктокутикулы. Эктокутикула пропитана дубильными веществами и обладает высокой прочностью. Эндокутикула во время линьки растворяется и всасывается гиподермой, а эктокутикула целиком сбрасывается. В состав хитиновой кутикулы входят разнообразные пигменты.

Нервная система представлена парным надглоточным узлом — головным мозгом, окологлоточным нервным кольцом и брюшной нервной цепочкой (у низших форм она в виде лестницы). От всех нервных

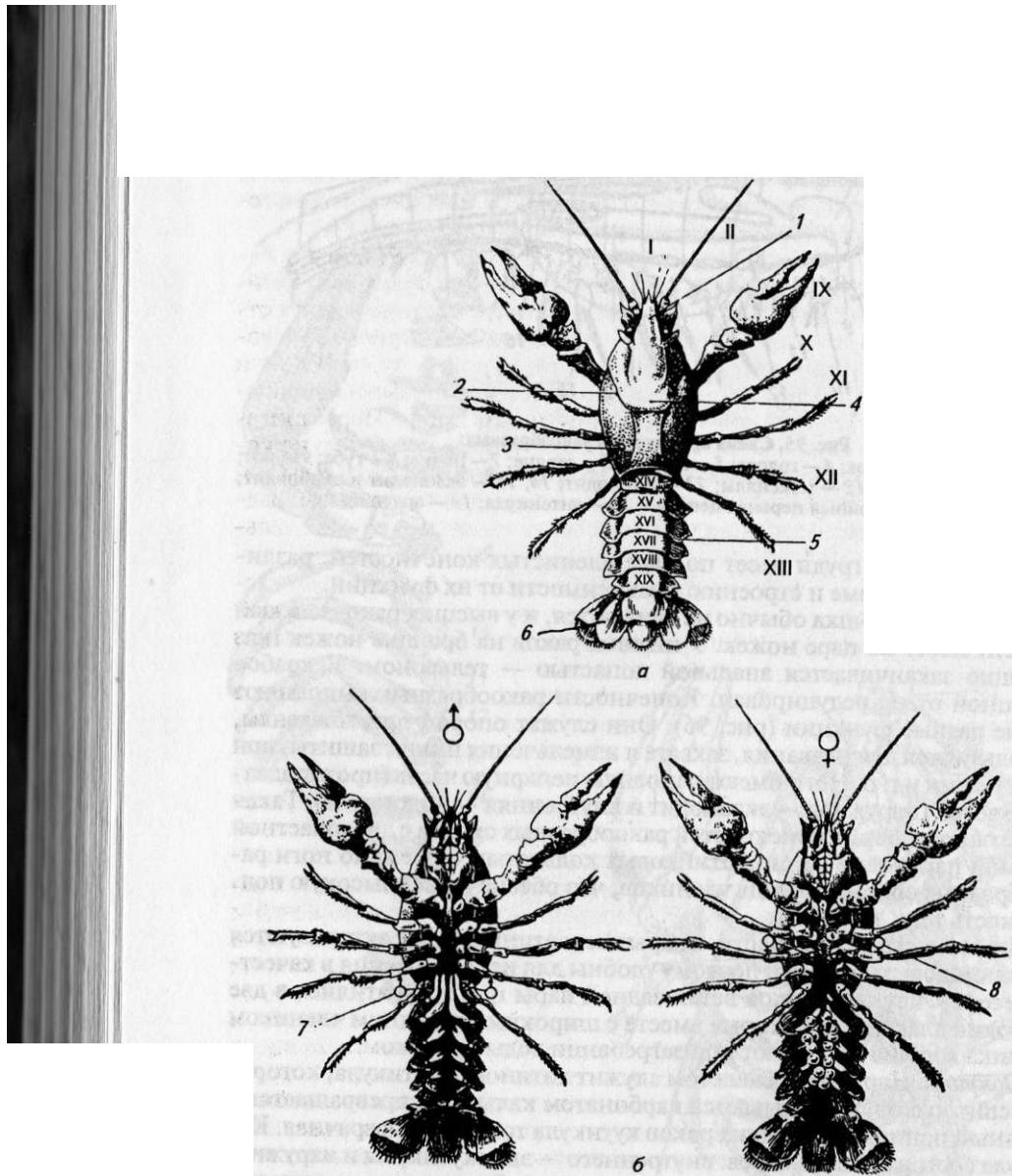


Рис. 94. Речной рак:
а — со спинной стороны; б — с брюшной стороны; 1 — рострум; 2 — головогрудной щит;
3 — края этого щита, покрывающие жабры; 4 — головогрудь; 5 — брюшко; 6 — хвостовой
плавник; 7 — мужское половое отверстие; 8 — женское половое отверстие; конечности и
сегменты: I — антеннула; II — антenna; IX—XIII — ходильные ноги; XIV—XIX — сегмен-
ты брюшка

у злов отходят периферические нервы. Органы чувств хорошо развиты: простые и сложные глаза, две пары антенн, органы равновесия. У глубоководных, сидячих и паразитических видов глаза могут отсутствовать. Органы осязания в виде чувствительных щетинок разбросаны по всему телу, но особенно их много на усиках. На первой паре антенн со средоточены хеморецепторы, здесь же находятся статоцисты.

Пищеварительная система. Эктодермальная передняя кишечная система представлена пищеводом, переходящим в желудок, который выстлан хитином (рис. 97). У некоторых высших раков желудок состоит из двух отделов: жевательного и пилорического. В жевательном отделе происходит измельчение пищи при помощи трех выступов — зубов. Стенки пилорического отдела желудка имеют складки с щетинками для процеживания мелкоизмельченной и жидкой фракций пищи. Переваривание и всасывание измельченной пищи происходит в относительно короткой эндодермальной кишке, куда открываются протоки печени, секрет которой выполняет функции сока поджелудочной железы. Задняя кишечка прямая, выстлана кутикулой и открывается наружу анальным отверстием.

Мелкие раки, ведущие планктонный образ жизни, захватывают частицы пищи с помощью усиков, ротовых конечностей, грудных ножек, создающих ток воды. У дафний задние грудные ножки бьют до 300 раз в 1 мин, обеспечивая постоянное поступление пищи в рот.

У ракообразных ротовые конечности выполняют разнообразные функции. Так, у раков хорошо развиты верхние челюсти — жвалы, имеющие зазубренный край для перетирания пищи. Две пары нижних челюстей также принимают участие в механической переработке пищи. Три пары ногочелюстей, расположенные на сегментах груди, удерживают добычу и подносят ее ко рту. Большие раки захватывают добычу первой парой ходильных ног, вооруженных мощными клемшами.

Органы дыхания. Большинство ракообразных дышат кожными жабрами, которые представляют собой придатки грудных конечностей и имеют вид тонких выростов у основания ножек. У высших раков жабры образуются не только на ногах, но и на стенке тела в жаберных полостях I юд хитиновым щитом — карапаксом. Движение воды около жабр происходит за счет движения ног, у основания которых они находятся, а также тех ножек, которые не имеют жабр. У сухопутных мокриц на брюшных ножках находятся глубокие ветвящиеся впчивания — псевдодрагеи, в которых происходит газообмен, требующий повышенной влажности воздуха (до 90 %). Мелкие низшие раки дышат поверхностью тела. Сухопутным крабам также необходима высокая влажность воздуха.

Кровеносная система. У высших раков кровеносная система представлена мешковидным и вытянутым вдоль спинной стороны тела сердцем, имеющим отверстия (остии), через которые из полости тела засасывается кровь (гемолимфа). От сердца по артериям кровь выливается в лакуны миксоцеля тела. В полости тела гемолимфа отдает кислород тканям и насыщается диоксидом углерода. Частично кровь

отдает конечные продукты обмена почкам. В крови ракообразных содержится гемоцианин или гемоглобин, связывающие кислород. Из полости тела венозная кровь собирается в систему венозных сосудов и по жаберным приносящим сосудам поступает в систему капилляров в жабрах. Здесь кровь освобождается от диоксида углерода и насыщается кислородом. По выносящим жаберным сосудам кровь поступает в пикоардиальный синус, окружающий сердце.

У некоторых раков кровеносная система представлена только сердцем или кровь перемещается за счет работы мышц тела или за счет движения кишечника. Иногда у мелких раков кровеносная система может полностью редуцироваться.

Органы выделения. В головном отделе располагается две пары почек — видоизмененных и более сложно устроенных метанефридиев. Выводные протоки первой пары почек открываются у основания второй пары антенн (антеннальные железы), второй пары почек — у основания второй пары нижних челюстей — максилл (максиллярные железы). Каждая почка состоит из мешочка и выделительного канальца, который, расширяясь, может образовывать мочевой пузырь. У большинства ракообразных имеется лишь одна из двух пар почек — антеннальные или максиллярные.

Органы размножения. Большинство ракообразных раздельнополы. Встречается половой диморфизм. У немногих сидячих форм и паразитов наблюдается гермафроптизм. Развитие с метаморфозом разной степени сложности, реже развитие протекает без стадии личинки. У низших раков из яиц выходит личинка науплиус (рис. 98), имеющая три пары ног и один глаз. У высших морских раков из яиц выходит ли-

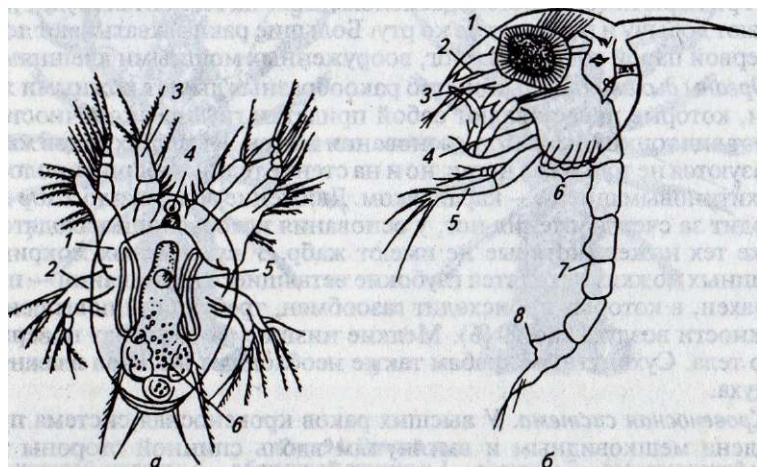


Рис. 98. Личинки раков:
 а — науплиус; 1,2,3 — конечности; 4 — простой глаз; 5 — орган выделения; 6 — кишечник; 6 — зоэа; 1 — сложный глаз; 2—5 — конечности; 6 — зарядки брюшных ног; 7 — брюшко; (У — последняя пара конечностей)

чинка зоэа, имеющая большее число сегментов и, следовательно, конечностей и два глаза. Кутикула личинок зоэа имеет шипики, увеличивающие ее поверхность и облегчающие личинкам плавание в толще воды.

Иногда у самок имеются семяприемники; некоторые самцы образуют сперматофоры, которые приклеивают к телу самок или вводят их в половые пути самок. У речного рака женские половые отверстия открываются на шестом грудном сегменте у основания третьей пары ходильных ног. Мужские половые отверстия расположены на восьмом грудном сегменте у основания пятой пары ходильных ног. У самцов две пары первых брюшных ножек превращены в копулятивные трубочки, с помощью которых самцы вводят сперму в половые отверстия самок.

Класс ракообразных делят на пять подклассов, из которых будут рассмотрены три: Жаброногие (Branchiopoda), Максиллоподы (Maxillipoda) и Высшие раки (Malacostraca).

ПОДКЛАСС ЖАБРОНОГИЕ (BRANCHIOPODA). К подклассу Жаброногие принадлежит 400 видов самых примитивных и мелких обитателей морей и пресных водоемов, у которых голова не срастается с грудными сегментами и отсутствуют конечности на брюшке. У жаброногих имеются сложные глаза и двуветвистые листовидные грудные ножки. Тельсон заканчивается вилочкой. Выделительные железы максиллярные. У основания ножек расположены жаберные лепестки; иногда жаброногие дышат через покровы тела. Развитие происходит с метаморфозом: из яйца вылупляется личинка науплиус, которая превращается во взрослую особь. Иногда развитие бывает прямым.

Наиболее типичными представителями жаброногих являются различные виды дафний (*Daphnia*) из подотряда Ветвистоусые раки (Cladocera) отряда Листоногие (Phyllopoda), в огромном количестве заселяющие пресные водоемы (рис. 99). Тело дафний мешкообразной формы и заключено в карапакс, напоминающий по форме двустворчатую тонкую хитиновую раковину. Створки карапакса приоткрыты с брюшной стороны. На голове имеется один фасеточный глаз и две пары антенн, из которых первая пара (антеннулы) небольших размеров. Вторая пара очень крупных антенн имеет двуветвистое строение и принимает участие в плавании. Взмахи антенн дают скачкообразные движения ветвистоусых, за что их называют водяными блохами.

Грудной отдел представлен четырьмя—шестью сегментами, несущими по паре коротких листовидных ножек, у основания которых расположены жаберные лепестки, а также ряды щетинок. Вода движется внутри створок раковины под действием ножек и омывает жабры. Пищевые частицы, увлекаемые током воды, отфильтровываются щетинками ног и отправляются в рот.

Наиболее распространена обыкновенная дафния (*Daphnia pulex*). В течение летнего периода дафнии размножаются партеногенетически, давая в потомстве только самок. Последние также дают партеногенетические поколения самок и т. д. Неоплодотворенные яйца откладывются в выводковую камеру, из них выходят молодые ракчи. Осенью при наступлении холодов самки откладывают порцию неоплодо-

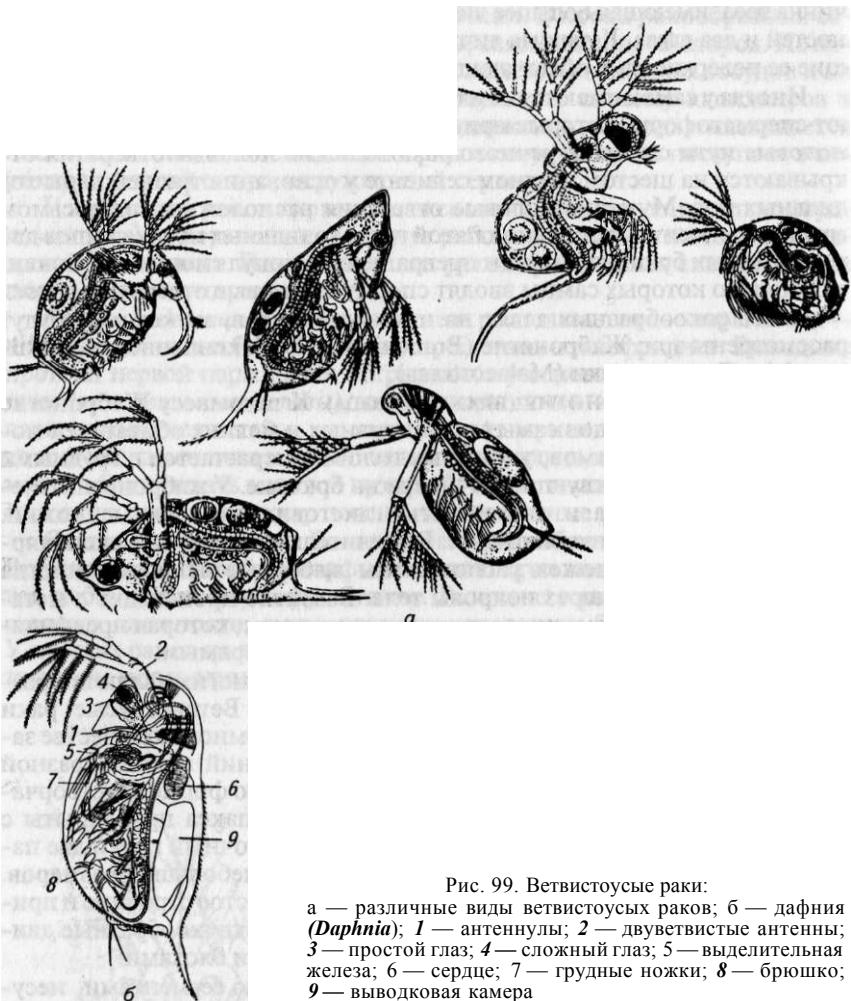


Рис. 99. Ветвистоусые раки:
а — различные виды ветвистоусых раков; б — дафния (*Daphnia*); 1 — antennулы; 2 — двуветвистые антennы; 3 — простой глаз; 4 — сложный глаз; 5 — выделительная железа; 6 — сердце; 7 — грудные ножки; 8 — брюшко; 9 — выводковая камера

творенных яиц, из которых развиваются только самцы. Эти же самки через некоторое время откладывают в камеру вторую порцию неоплодотворенных яиц, которые претерпевают деление и получают гаплоидный набор хромосом. Эти яйца оплодотворяются молодыми самцами и в выводковой камере покрываются плотной оболочкой, образуя эфиопий (в каждом по одному-два яйца). Эфиопии зимуют, а весной из них снова появляются самки, которые дают новое поколение партеногенетических самок. Таким образом, у дафний происходит чередование партеногенетических и полового поколений, т. е. жизненный цикл протекает по типу гетерогонии.

ПОДКЛАСС МАКСИЛЛОПОДЫ (MAXILLOPODA). Представители подкласса Максиллоподы имеют упрощенную организацию: у них образуется головогрудь, дышат они поверхностью тела, у большинства нет сердца и сосудов. Мандибулы массивные. Максиллы представляют цедильный аппарат. Брюшных ножек нет. Мелкие ракчи со стройным удлиненным телом циклопы (*Cyclops*) и диаптомусы (*Diaptomus*) из отряда Веслоногие (Copepoda) движутся вперед с помощью двуветвистых грудных ножек, которые уплощены и снабжены щетинками. На голове расположены развитая первая пара одноветвистых антеннул и пара коротких антенн (рис. 100). Брюшко заканчивается вилочкой.

Циклопы и диаптомусы заселяют в большом количестве разнообразные водоемы и служат кормом для рыб и других обитателей водоемов. Размножаются эти ракообразные только половым путем. Оплодотворенные яйца самки склеивают в два яйцевых мешка, которые прикрепляют к нижней стороне первого брюшного сегмента. Развитие происходит со стадией науплиуса.

Известно около 1,8 тыс. видов веслоногих, живущих в морях и пресных водоемах. В морском планктоне наиболее многочисленны каланусы (*Calanus*). Они служат кормом для рыб и китов. Среди циклопов много фильтраторов, фитофагов и хищников. Яйца циклопов очень устойчивы к неблагоприятным условиям среды. Есть среди веслоногих ракообразных эктопаразиты рыб, некоторые являются промежуточными хозяевами плоских червей.

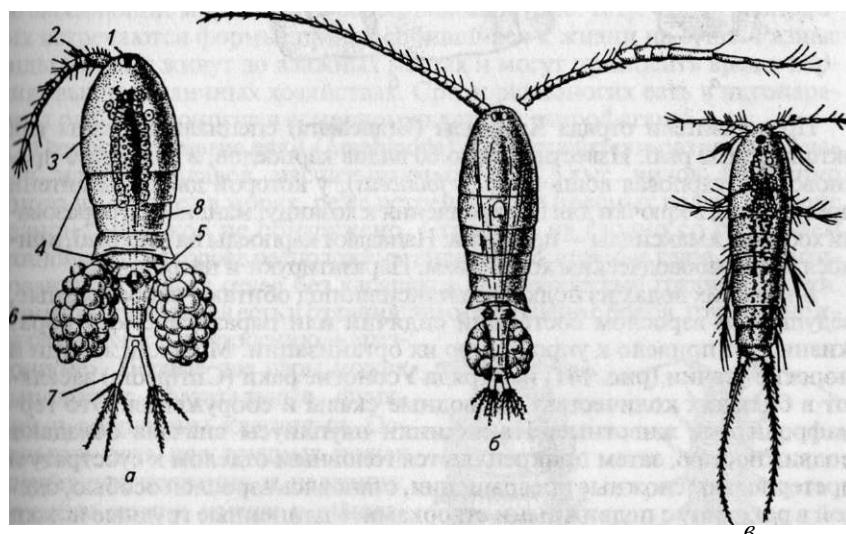
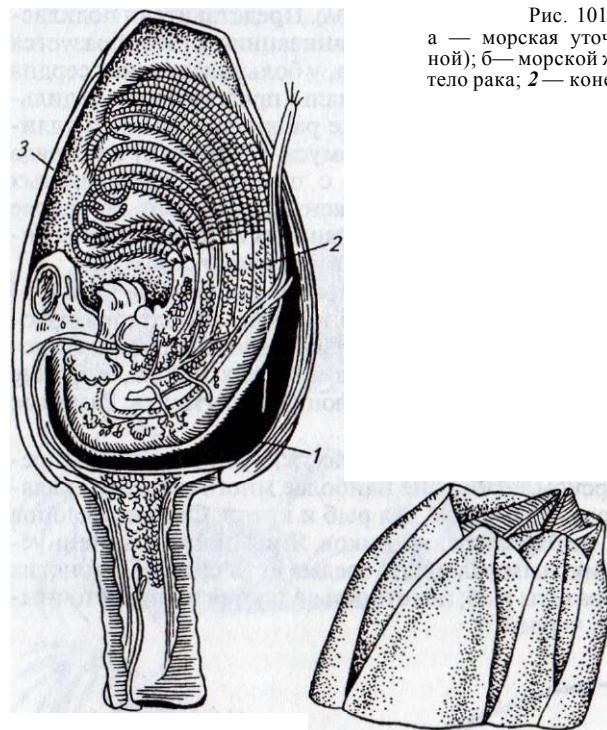


Рис. 100. Свободноживущие веслоногие ракообразные:
а - циклоп; б - диаптомус; в - кантокамптус; 1 — непарный глаз; 2 — первая пара антенн; 3 — головогрудь; 4 — свободные сегменты груди; 5 — брюшко; 6 — яйцевые мешки; 7 — вилочка брюшка; 8 — кишечник

Рис. 101. Усоногие раки:
а — морская уточка (со вскрытой раковиной); б— морской желудь (внешний вид); 1 —
тело рака; 2 — конечности; 3 — раковина



Представители отряда **Карпоеды (Branchiura)** специализированы как эктопаразиты рыб. Известно около 60 видов карпоедов, в том числе пресноводная карповая вошь (*Argulus foliaceus*), у которой две пары антенн превращены в крючки для прикрепления к хозяину, мандибулы образовали хоботок, а максиллы — присоски. Нападают карпоеды на всех рыб, принося вред рыбоводческим хозяйствам. Паразитируют и на лягушках.

В морских водах из подкласса максиллопод обитают ракообразные, ведущие во взрослом состоянии сидячий или паразитический образ жизни, что привело к упрощению их организации. Морские желуди и морские уточки (рис. 101) из отряда Усоногие раки (Cirripedia) заселяют в больших количествах подводные скалы и сооружения. Это гермафродитные животные. Их личинки науплиусы сначала обладают подвижностью, затем прикрепляются головным отделом к субстрату и претерпевают сложные превращения, становясь взрослой особью, одетой в раковину с подвижными створками. Удлиненные грудные ножки служат для передвижения воды в раковине, способствуя дыханию и процеживанию пищевых частиц. Наиболее многочисленны во всех морях морские желуди (*Balanus*) и морские уточки (*Lepas*). Они обрастают суда и подводные сооружения.

ПОДКЛАСС ВЫСШИЕ РАКИ (MALACOSTRACA). К этому классу принадлежат ракообразные средних и крупных размеров, населяющие морские и пресные водоемы; некоторые приспособились к жизни на суше. Их тело характеризуется постоянством сегментарного состава и состоит из акрона и четырех головных сегментов, восьми грудных и шести-семи брюшных сегментов. Часто головные и грудные сегменты сливаются, образуя головогрудь. Конечности имеются на всех сегментах, в том числе и на брюшных. Развитие происходит без превращения или со стадией зоэа. У низших раков развитие происходит со стадией науплиуса. Личинка зоэа в отличие от науплиуса имеет удлиненное и расчлененное тело. Рост раков сопровождается линьками. Так, речной рак в первый год жизни линяет около десяти раз, в течение второго года жизни число линек снижается до пяти, а на третий год жизни происходит всего две линьки. В последующем самцы линяют дважды, а самки — лишь раз в год. Практически рост раков прекращается к 5 годам, живут они до 20 лет.

Из 26 тыс. видов высших раков наибольший интерес представляют три отряда: Равноногие, Разноногие и Десятиногие раки.

Отряд Равноногие раки (Isopoda). В отряде 4,5 тыс. видов небольших раков с уплощенным телом. Передний грудной сегмент сливается с головным отделом. Глаза фасеточные. Карапакса нет. Сегменты груди и брюшка несут по паре коротких ножек. Ходильные грудные ножки одноветвистые и одинакового строения, что и определило их название «равноногие». Брюшные ножки двуветвистые и выполняют дыхательную функцию. К отряду равноногих относятся водяные ослики, обильно заселяющие морские и пресные водоемы (рис. 102). Среди равноногих встречаются формы, приспособившиеся к жизни на суше. Разные виды мокриц живут во влажных местах и могутносить вред в парниковых и тепличных хозяйствах. Среди равноногих есть и эктопаразиты рыб. Равноногие в основном являются сапрофагами.

Отряд Разноногие раки (Amphipoda). Представители разноногих раков, или бокоплавов, насчитывают около 4,5 тыс. видов. Особенно много их обитает в морях, реже встречаются в пресных водах. На суше разноногих раков не обнаружено. Строение их сходно со строением равноногих: на голове расположены сидячие фасеточные глаза, сегментированный грудной отдел без карапакса, одноветвистые грудные ножки. (Однако у бокоплавов есть и отличия: тело сплющено с боков, грудные ножки различаются по строению — «разноногие». Первые две пары ножек выполняют хватательные функции и вооружены клешнями. Остальные пять пар грудных ножек служат для ползания и плавания. Грудные ножки имеют у основания

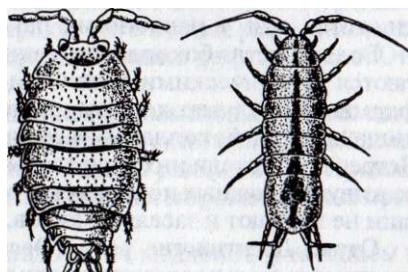


Рис. 102. Равноногие раки:
ч — мокрица; б — водяной ослик

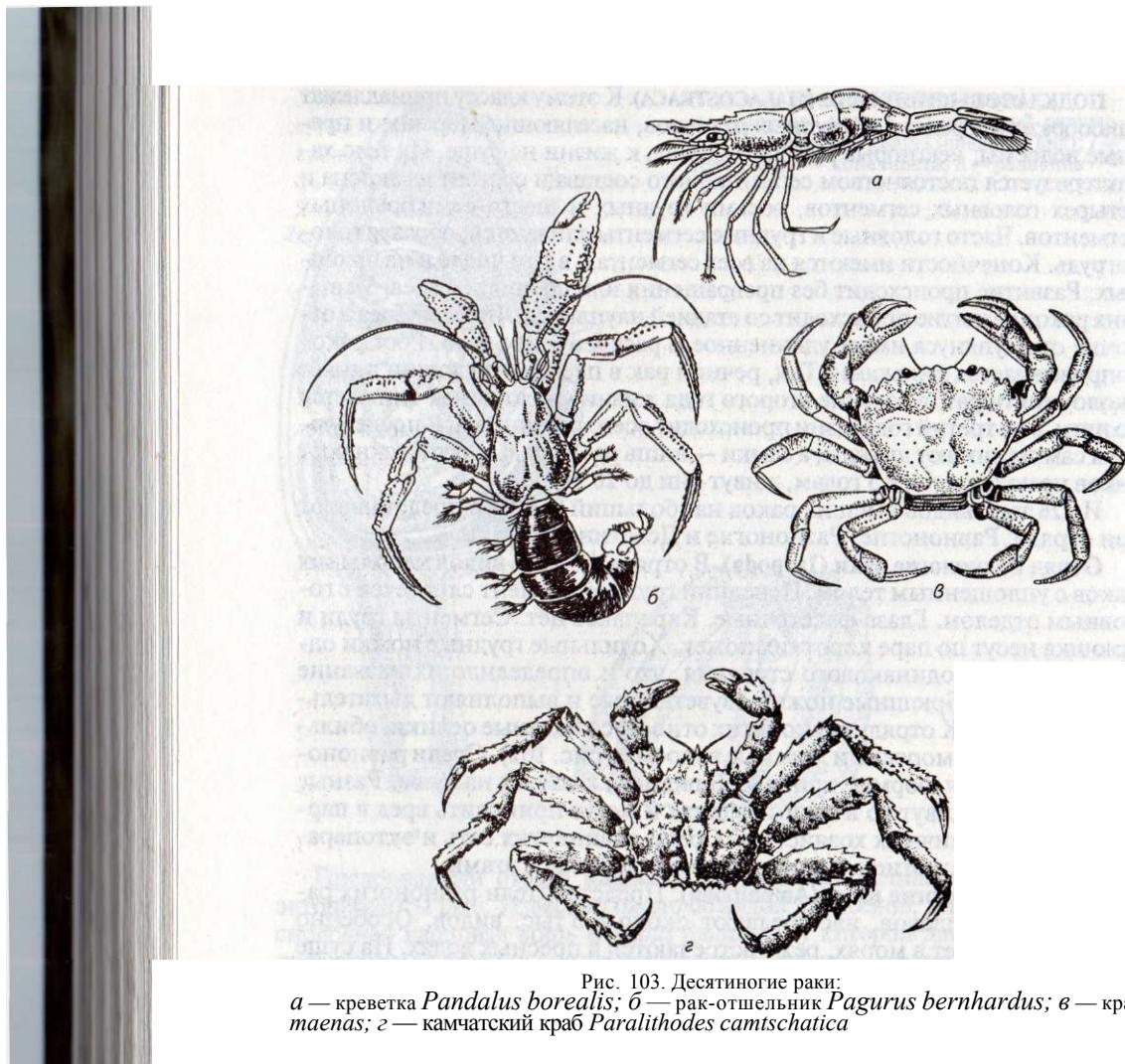


Рис. 103. Десятиногие раки:
а — креветка *Pandalus borealis*; б — рак-отшельник *Pagurus bernhardus*; в — краб *Carcinus maenas*; г — камчатский краб *Paralithodes camtschatica*

ния жаберный аппарат, поэтому сердце у бокоплавов находится в груди, тогда как у равноногих раков оно лежит в брюшном отделе.

Большинство бокоплавов — придонные ракообразные, которые питаются органическими остатками. Есть пелагические и планктонные формы. Они служат кормом для рыб. Во вновь созданные водохранилища и озера бокоплавов заселяют для укрепления кормовой базы. Встречаются среди них и паразиты: китовые вши (сем. Cyamidae) в массе живут на кожных покровах китов, вызывая их изъязвления. Китовые вши не плавают и заселяют китов, переползая с одного на другого.

Отряд Десятиногие раки (Decapoda). К ним относятся наиболее крупные и высокоорганизованные ракообразные (рис. 103), заселяю-

щие в основном моря и реже пресные воды, есть сухопутные формы. Насчитывают 8,5 тыс. видов, у которых сегменты головы и груди слиты в головогрудь, прикрытую с боков и сверху головогрудным хитиновым щитом — карапаксом. Хитин щита пропитан известью. Из восьми пар грудных конечностей три передние участвуют в захвате пищи; это так называемые ногочелюстии. Остальные пять пар — ходильные ноги, с их помощью раки передвигаются. Часто передняя пара ходильных ног заканчивается мощными клешнями. Брюшко представлено шестью сегментами, каждый из которых несет по паре ног.

Жабры располагаются не только на всех грудных ногах, но и на теле у основания ног. Карапакс закрывает тело с боков, образуя жаберные крышки. На брюшке имеется хвостовой плавник, образованный последним сегментом тела и широкими лопастями ножек предпоследнего сегмента. У некоторых десятиногих раков брюшко редуцировано. Развитие прямое или с метаморфозом.

Отряд десятиногих раков подразделяется на два подотряда: подотряд Плавающие раки (*Natantia*) и подотряд Ползающие раки (*Reptantia*). Зачастую десятиногих раков делят на три группы: длиннохвостые, мягкохвостые и короткохвостые.

Подотряд Плавающие раки (*Natantia*) включает наиболее примитивную группу десятиногих раков, ведущих плавающий образ жизни. Типичными представителями плавающих раков являются разнообразные креветки (*Pandalus*, *Crangon* и др.): тело их сплющено с боков, брюшко длинное и несет плавательные ножки. Грудные ножки тонкие и без клешней, с их помощью креветки плавают, дышат и захватывают пищу. Креветки являются объектом промысла. Сами они служат кормом для морских обитателей, особенно для рыб.

Подотряд Ползающие раки (*Reptantia*) — это прогрессивная группа десятиногих раков, в основном ведущих хищнический образ жизни, захватывая добычу клешнями. Брюшные ножки не выполняют плавательной функции и развиты слабо. У части раков брюшко существенно редуцировано (крабы). У представителей этого подотряда явно выраженная тенденция к передвижению ползанием, тогда как способность к плаванию существенно снижается. Подотряд подразделяется на отдельные: лангусты (*Palinura*), омары (*Astacura*), крабы (*Brachyura*) и отшельники, или крабоиды (*Anomura*). У омаров и лангустов брюшко мускулистое и длинное, они могут плавать. У отшельников брюшко асимметрично и недоразвито, раки прячут его в пустые раковины моллюсков или подгибают под себя. У крабов брюшко редуцировано. Отшельники и крабы плавать не могут.

Среди ползающих раков много ценных промысловых видов: лангуст (*Palinurus*), омар (*Homarus*), крабоид камчатский краб (*Paralithodes camtschatica*), крабы (*Cancer*, *Callinectes*), речной рак (*Astacus*). Промысел десятиногих раков широко развит: ежегодная мировая добыча приближается к 1 млн т.

Речные раки обитают в пресных водоемах с медленным течением и чистой водой. Ведут ночной и сумеречный образ жизни, питаясь дон-

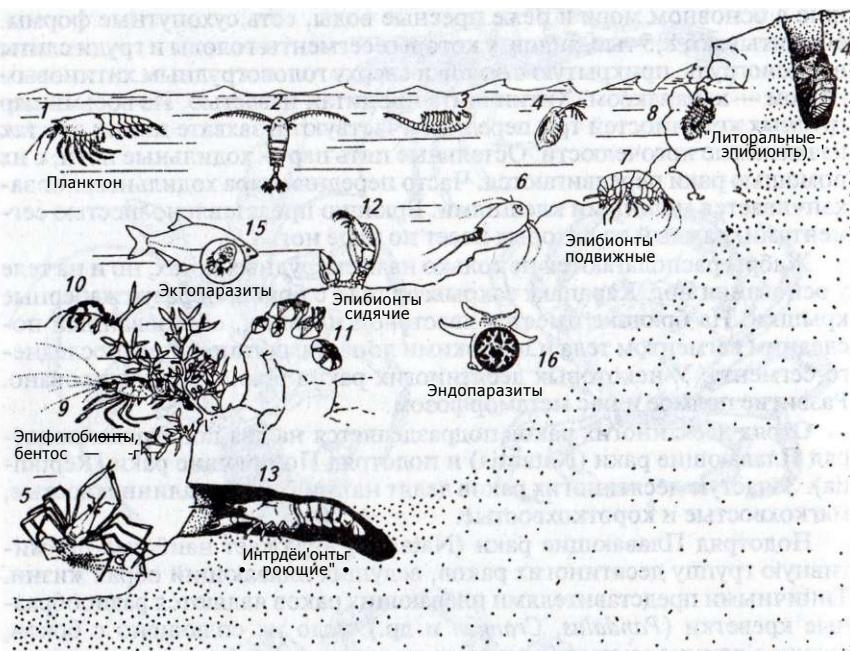


Рис. 104. Экологическая радиация ракообразных:
 1 — креветка; 2 — циклоп; 3 — жабромог; 4 — дафния; 5 — краб; 6 — щитень; 7 — бокоплав;
 8 — равноногий рак; 9 — креветка; 10 — равноногий рак; 11 — морские жемуди; 12 — морские
 уточки; 13 — десятиногий рак; 14 — роющая мокрица; 15 — карповая вошь; 16 — саккулина

ными животными и падалью. Самки вынашивают оплодотворенную икру, прикрепляя ее к ножкам брюшка. Речные раки и морские раки являются объектами промысла, а речного рака разводят в специализированных хозяйствах.

Филогения ракообразных. Ракообразные являются древней группой членистоногих животных. Предполагается, что каждый подкласс ракообразных имеет собственную линию развития от общих предков. Эти гипотетические предки обладали целым комплексом исходных признаков, которые проявляются у наиболее примитивных форм современных раков. Каждый из современных подклассов имеет эти признаки. У жаброногов Branchiopoda голова неслитная, брюшная нервная цепочка лестничного типа, сердце трубчатое. У представителей Maxillopoda примитивны двуветвистые головные конечности — антенны и челюсти, которыми они пользуются при передвижении. Только подкласс Ostracoda в своем эволюционном развитии утратил почти все примитивные признаки, и его современные представители практически лишиены их.

Таким образом, все подклассы ракообразных представляют потомков древней предковой группы Crustacea. Эволюция ракообразных

привела к образованию различных жизненных форм, занявших разнообразные экологические ниши (рис. 104). Исходным типом, видимо, были мелкие пелаго-бентосные формы, которые вели плавающий образ жизни. От этих форм специализация шла в нескольких направлениях: планктон, некton и бентос. Часть представителей приспособилась к паразитизму, а некоторые группы вышли на сушу, где положительные температуры и высокая влажность позволили им вести активный образ жизни без существенной перестройки органов дыхания.

ПОДТИП ХЕЛИЦЕРОВЫЕ (*Chelicerata*)

Хелицеровые — это особая ветвь членистоногих, по своим морфологическим характеристикам обособленная от других подтипов. Насчитывают около 63 тыс. видов современных хелицеровых — обитателей суши, представленных в основном паукообразными. Среди паукообразных встречаются вторичноводные виды, паразиты растений и животных. Для многих характерно выделение паутинных нитей из особых паутинных желез. Паутина помогает паукообразным в защите от врагов, в добывче пищи, расселении и т. п.

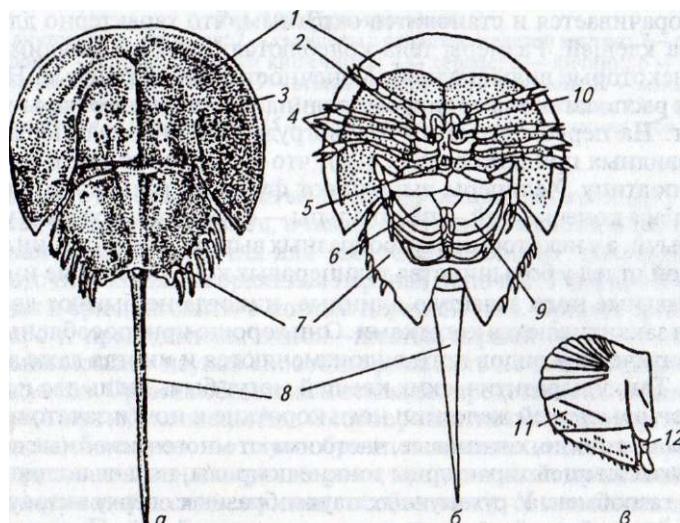


Рис. 105. Строение мечехвоста:
а — вид со спинной стороны; б — вид с брюшной стороны; в — жаберная ножка; 1 — головогрудной щит; 2 — рот; 3 — сложные глаза; 4 — ходильные ноги; 5 — хилярий; 6 — жаберная крышка; 7 — брюшной отдел; 8 — мечевидный отросток; 9 — жаброносные ножки; 10 — хелидеры; 11 — жаберный придаток; 12 — членистая ножка

Наиболее часто у хелицеровых сливаются сегменты головы и груди, образуя головогрудь. Это сильнее всего выражено у клещей. Антенные (усики) отсутствуют. Глаза простые, от одной до восьми пар. Функции усиков и челюстей выполняют первые две пары членистых пришатков: хелицыры и педипальпы. Имеется четыре пары ходильных ног. На брюшке обычно конечностей нет, у части сухопутных видов они видоизменены в половые пришатки, органы дыхания или в паутинные бородавки. Подтип Хелицеровые включает два класса: Мечехвосты (*Xiphosura*) и Паукообразные (*Arachnida*).

Мечехвосты представлены всего пятью ныне живущими видами (рис. 105), хотя в древности это была большая и широко распространенная группа водных животных. Мечехвосты имеют уплощенную головогрудь, закрытую панцирем, и широкое слитное брюшко, которое заканчивается мечевидным отростком. Ведут бентосный роющий образ жизни.

КЛАСС ПАУКООБРАЗНЫЕ (*Arachnida*)

Строение и жизненные отправления. Форма тела паукообразных весьма разнообразна. Тело состоит из головогруди и брюшка. Брюшко сегментированное, реже слитное, число сегментов достигает 12, и заканчивается брюшко тельсоном. У сильно расчлененных паукообразных (скорпионы, сольпуги) тело вытянутое, по мере слияния сегментов тело укорачивается и становится округлым, что характерно для большинства клещей. Размеры тела колеблются от долей миллиметра до 20 см (некоторые виды пауков). Конечности одноветвистые. На головогруди располагаются хелицыры, педипальпы и четыре пары ходильных ног. На первом сегменте головогруди имеются хелицыры в виде клешневидных пришатков (рис. 106), что и дало соответствующее название подтипу. Хелицыры выполняют функции размельчения пищи. Вторая пара конечностей — педипальпы — служат для захвата и удержания добычи, а у некоторых паукообразных выполняют функции антенн. Брюшной отдел у большинства хелицеровых конечностей не имеет.

Ходильные ноги зачастую длинные, никогда не бывают двуветвистыми и заканчиваются коготками. Они хорошо приспособлены к бегу. У паразитических видов ноги видоизменяются и иногда даже атрофируются. Так, у паразитических клещей могут быть лишь две передние пары ног, а у клещей железниц ноги короткие и почти зачаточные.

Покровы тонкие, хитиновые, часто имеют многочисленные волоски. Для мелких клещей характерны тонкие покровы, позволяющие осуществлять газообмен. У сухопутных паукообразных сверху экзокутикулы имеется тонкий слой эпикутикулы, в состав которой входят воскоподобные вещества, но нет хитина. Этот слой хорошо защищает тело от высыхания. В целом кутикула у паукообразных не бывает толстой. Лишь у вторичноводных представителей кутикула превращается в панцирь. У панцирных клещей кутикула также утолщается. К производным наруж-

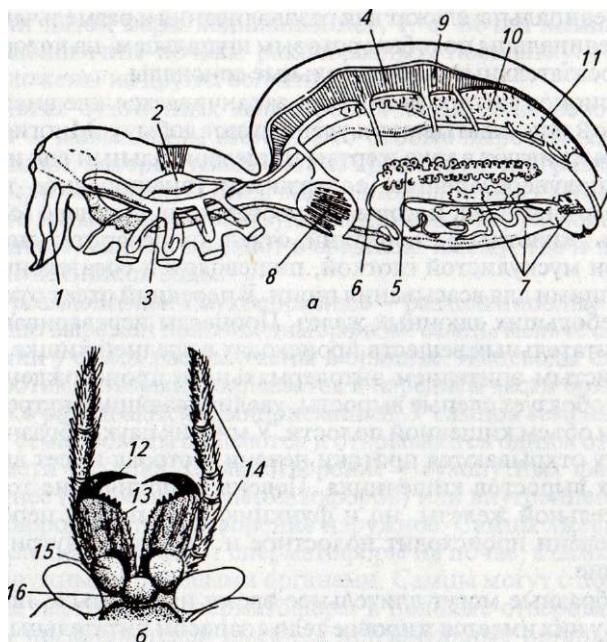


Рис. 106. Паук-крестовик:
а — схема внутреннего строения; 1 — хелицеры с ядовитой железой внутри; 2 — сосательный желудок; 3 — слепые отростки кишечника; 4 — сердце; 5 — яичник; 6 — яйцевод; 7 — паутинные железы; 8 — легкое; 9 — отверстие сердца; 10 — печень; 11 — мальпигиевые сосуды; б — хелицеры и педипальпы; 12 — когтевидный членник хелицер; 13 — основной членник хелицер; 14 — педипальпы; 15 — челюстная лопасть; 16 — нижняя губа

пого покрова относятся ядовитые железы у основания хелицер у пауков и ядовитая игла у скорпионов, а также паутинные железы пауков.

Нервная система типична для всех членистоногих: головной мозг, окологлоточное кольцо и брюшная нервная цепочка. У пауков и клещей узлы груди и брюшка слиты в единый нервный узел. Органы зрения развиты слабо и представлены одной—шестью парами простых глаз. Два центральных глаза у пауков способны различать не только форму, но и цвет предметов. Органы обоняния и осязания представлены отдельными сенсиллами или их скоплениями. Особое развитие получили органы осязания, так как большинство паукообразных ночные хищники и зрение не может играть в их жизни такую же роль, как у дневных представителей.

Пищеварительная система имеет особенности, связанные с характером питания. Паукообразные имеют две пары ротовых конечностей: хелицеры — челюсти и педипальпы — ногощупальца. Хелицеры состоит из основного членика и остального, загнутого крючка, который пауки вонзают в тело жертвы (см. рис. 106). Педипальпы состоят из основного членика, с которым соединен членистый щупик. У хищных пауков хе-

лидеры и педипальпы служат для раздавливания и размельчения пищи. У пауков педипальпы подобны ротовым щупальцам, на которых сосредоточены осязательные и обонятельные сенсиллы.

У скорпионов щупик педипальп заканчивается клешней, с помощью которой они схватывают и удерживают добычу. Многие паукообразные впрыскивают в тело жертвы пищеварительный сок и затем всасывают полупереваренное содержимое (внекишечное пищеварение). У представителей, которые питаются жидкой пищей (соки растений, кровь животных), передний отдел пищеварительного тракта представлен мускулистой глоткой, пищеводом и сосательным желудком, служащими для всасывания пищи. В передний отдел открываются протоки небольших слюнных желез. Процессы переваривания и всасывания питательных веществ происходят в средней кишке, выстланной железистым эпителием энтодермального происхождения. Средняя кишка образует слепые выросты, увеличивающие внутреннюю поверхность и объем кишечной полости. У многих паукообразных в среднюю кишку открываются протоки печени, которая имеет вид парных железистых выростов кишечника. Печень выполняет не только роль пищеварительной железы, но и функцию всасывания переваренной пищи. В печени происходит полостное и частично внутриклеточное пищеварение.

Паукообразные могут длительное время не питаться, поскольку в миксоцеле у них имеется жировое тело с запасом питательных веществ.

Кровеносная система представлена лежащим на спинной стороне тела мускулистым сердцем и отходящими от него сосудами, которые идут к различным органам. Обратный ток крови осуществляется по лакунам. У клещей часть кровеносных сосудов редуцирована, иногда отсутствует и сердце. У паукообразных, имеющих легочные мешки, кровеносная система развита лучше, чем у трахейнодышащих.

Органы дыхания. У одних паукообразных это легочные мешки, у других — трахеи, у третьих — легочные мешки и трахеи одновременно (большинство пауков). У водных хелицеровых органы дыхания представлены жабрами. У некоторых мелких форм газообмен осуществляется через покровы тела. Легочные мешки расположены в передней части брюшка и открываются наружу специальными отверстиями — дыхальцами. В легочных мешках кровь течет в параллельно расположенных тонких листовидных складках, через которые и происходит газообмен: в щелевидные пространства между складками проникает воздух, отдающий кислород в гемолимфу сосудов, которые пронизывают листовидные складки легочных мешков.

Трахеи — наиболее распространенные органы дыхания у паукообразных — начинаются отверстиями, или стигмами, в покровах брюшка; от стигм в глубь тела расходятся трахеи в виде ветвящихся трубочек. Трахеи и легкие паукообразных возникли независимо друг от друга. Легочные мешки более древние органы, чем трахеи.

Органы выделения представлены коксальными железами (почками), которые открываются выделительными отверстиями у основания

третьей или пятой пары ходильных ног, т. е. почки хелицеровых не вполне гомологичны почкам ракообразных, поскольку у последних они расположены на других сегментах тела.

Для многих сухопутных хелицеровых характерны особые органы выделения — *мальпигиевы сосуды*. Это особые выросты задней части средней кишki, которые извлекают из крови продукты распада и отводят их в среднюю кишку. Они в виде одной-двух пар слепых трубочек небольшого диаметра способствуют рациональному расходованию воды в организме, так как впадают в среднюю кишку, где и происходит всасывание излишков воды.

Органы размножения. Паукообразные — раздельнополые животные; у них хорошо выражен половой диморфизм: самцы мельче самок. Партеногенные яичники у самок расположены в брюшке. Яйцеводы сливаются в единый проток, который открывается в передней части брюшка. У самок нередко развиваются семяприемники. У самцов семенники лежат в брюшке, семяпроводы сливаются и открываются одним отверстием в нижней части брюшка. Оплодотворение у сухопутных форм наружновнутреннее (с помощью сперматофоров) или внутреннее. У водных форм хелицеровых оплодотворение наружное. Самцы лжескорпионов и многих клещей оставляют сперматофоры на почве, а самки захватывают их наружными половыми органами. Самцы могут с помощью хелицер сами переносить сперматофоры вальные отверстия самок. У некоторых представителей имеются копулятивные органы, и в этом случае сперматофоры у самцов не образуются.

Большинство паукообразных откладывают яйца, у некоторых наблюдается живорождение. Чаще всего развитие происходит без метаморфоза и сопровождается ростом и неоднократными линьками. У клещей иногда наблюдается партеногенетическое размножение, а развитие происходит со стадией личинки.

Класс паукообразных подразделяется на множество отрядов, из которых будут рассмотрены наиболее интересные для сельскохозяйственных работников отряды: Скорпионы (*Scorpiones*), Ложноскорпионы (*Pseudoscorpiones*), Сольпуги, или Фаланги (*Solifugae*), Сенокосцы (*Opiliones*), Пауко! (*Aranei*) и три отряда клещей (*Acariformes*, *Parasitiformes*, *Opiliocarina*).

Отряд Скорпионы (*Scorpiones*). Это наиболее древние по происхождению паукообразные. Длина тела у тропических видов может достигать 18 см. Для скорпионов характерна наибольшая расчлененность тела. За головогрудью следует 6-сегментное переднебрюшье и 6-сегментное заднебрюшье (рис. 107). Дышат скорпионы легочными мешками. Тельсон образует вздутие с ядовитой иглой, на вершине которой открываются протоки ядовитых желез. Крупные педипальпы вооружены клешнями. Жертву скорпион захватывает клешнями педипальп, перегибает брюшко через спину вперед и вонзает иглу в добычу. Четыре пары ходильных ног заканчиваются парой коготков.

Известно около 600 видов скорпионов, обитающих в странах с теплым климатом. Этоочные хищники, днем они скрываются в расщепах

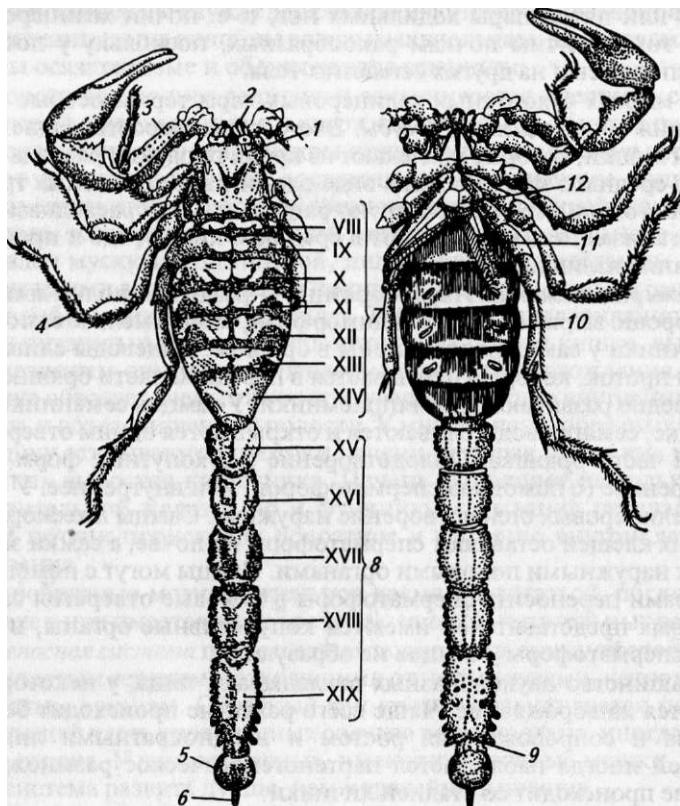


Рис. 107. Скорпион:
а — вид с спинной стороны; 1 — головогрудь; 2 — хелицеры; 3 — педипальпы; 4 — ноги;
5 — конечный членок брюшка; 6 — жало; 7 — передний отдел брюшка; 8 — задний отдел
брюшка; (VIII—XIX — сегменты брюшка); б — вид с брюшной стороны; 9 — анальное от-
верстие; 10 — легочная щель; 11 — гребенчатые органы; 12 — половые крылышки

линах и норках. Самки рождают детенышей и первое время носят их на спине. Укусы болезненны, но обычно не опасны для человека. В Закавказье обитает пестрый скорпион (*Buthus eureus*).

Отряд Ложноскорпионы (Pseudoscorpiones). Это мелкие (1—7 мм) паукообразные, имеющие клешневидные педипальпы и поэтому напоминающие скорпионов (рис. 108). Брюшко не разделено на передне- и заднебрюшье. Дышат ложноскорпионы с помощью трахей. На хелице-рах открываются протоки паутинных желез. Живут ложноскорпионы в лесной подстилке, под камнями и корой пней, в жилище человека. Питаются мелкими клещами. Самцы откладывают сперматофоры, которые захватываются самками в семяприемники. Яйца самка откладывает

ет в выводковую камеру, расположенную на брюшной стороне тела. Вышедшие из яиц личинки остаются подвешенными к камере и питаются желтком, который выделяет самка. После первой линьки молочные покидают мать.

Известно около 1,3 тыс. видов ложноскорпионов. В жилищах человека часто встречается книжный ложноскорпион (*Chelifer cancroides*), пытающийся мелкими насекомыми и клещами, которые вредят книгам.

Отряд Сольпуги, или Фаланги (Solifugae). Это крупные и сильно расчлененные обитатели степных и пустынных районов, насчитывающие около 600 видов. Головогрудь сольпуг неслитная и состоит из головного отдела и трех свободных сегментов, из которых последний недоразвит (см. рис. 108). Брюшко состоит из десяти сегментов. Педипальпы

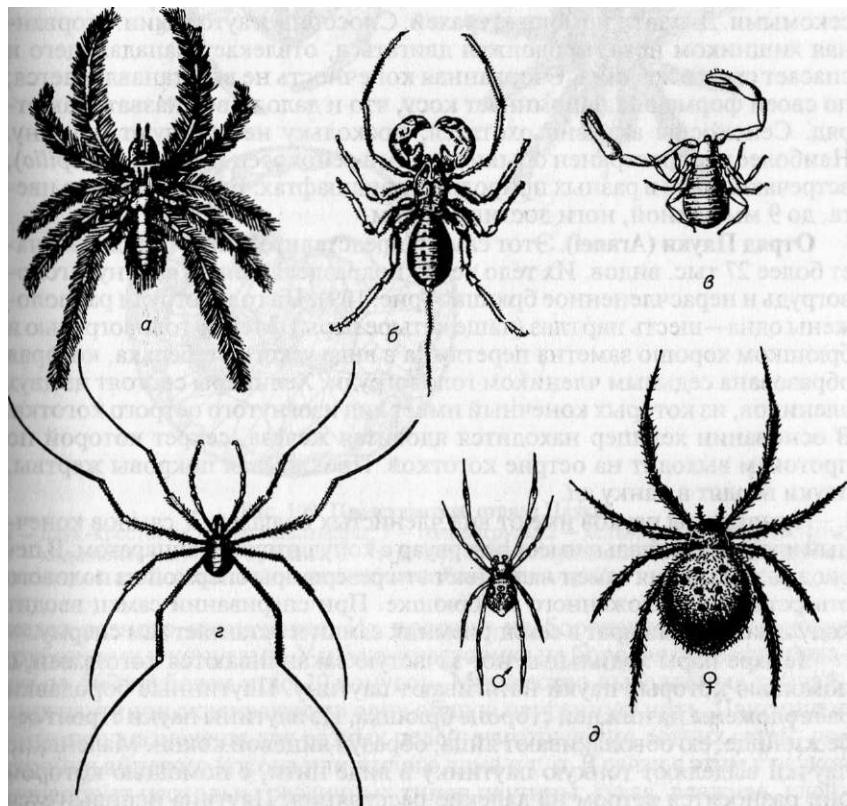


Рис. 108. Различные паукообразные:
а — сольпуга *Galeodes araneoides*; б — хвостатый телифон *Thelyphonus caudatus*; в — книжный лжескорпион *Chelifer cancroides*; г — обычный сенокося *Phalangium opilio*;
д — караукрут *Latrodectus tredecimguttatus*

похожи на ходильные ноги, участвуют в передвижении и выполняют чувствующие функции. Дышат с помощью трахей. Паутинных бородавок нет. Сольпуги питаются насекомыми, в том числе вредными (саранча), не ядовиты. Ведут ночной образ жизни. На Кавказе обитает фаланга *Galeodes araneoides*, имеющая в длину до 5 см. Самка откладывает яйца в норку и проявляет заботу о потомстве. Самец откладывает сперматофоры.

Отряд Сенокосцы (Opiliones) включает более 3 тыс. видов широко распространенных членистоногих, внешне похожих на пауков. Сенокосцы отличаются от пауков отсутствием перетяжки между головогрудью и брюшком, членистостью брюшного отдела, в котором десять сегментов, и клешневидными, а не крючковидными (как у пауков) хелицерами (см. рис. 108). Тело сенокосцев покоится на очень длинных и тонких ногах.

Сенокосцы повсеместно обитают на поверхности почвы, в трещинах коры деревьев, стенах строений и т. п., питаясь ночью мелкими насекомыми. Дышат с помощью трахей. Способны к аутотомии: оторванная хищником нога, продолжая двигаться, отвлекает нападающего и спасает своего хозяина. Оторванная конечность не восстанавливается; по своей форме она напоминает косу, что и дало повод назвать так отряд. Сенокосцы активно охотятся, поскольку не образуют паутину. Наиболее распространен обыкновенный сенокосец (*Phalangium opilio*), встречающийся в разных природных ландшафтах. Его тело бурого цвета, до 9 мм длиной, ноги достигают 5 см.

Отряд Пауки (Aranei). Этот самый представительный отряд включает более 27 тыс. видов. Их тело четко подразделяется на слитную головогрудь и нерасчлененное брюшко (рис. 109). На головогруди расположены одна—шесть пар глаз (чаще четыре пары). Между головогрудью и брюшком хорошо заметна перетяжка в виде узкого стебелька, которая образована седьмым члеником головогруди. Хелицеры состоят из двух члеников, из которых конечный имеет вид изогнутого острого коготка. В основании хелицер находится ядовитая железа, секрет которой по протокам выходит на острие коготков. Прокалывая покровы жертвы, пауки вводят в ранку яд.

Педипальпы пауков имеют вид членистых щупалец. У самцов конечный членик педипальп имеет резервуар с копулятивным аппаратом. В период размножения самец наполняет эти резервуары спермой из полового отверстия, расположенного на брюшке. При спаривании самец вводит копулятивный аппарат в семяприемник самки и оставляет там сперму.

Четыре пары ходильных ног зачастую заканчиваются коготками, с помощью которых пауки натягивают паутину. Паутинные бородавки расположены на нижней стороне брюшка. Из паутины пауки строят себе жилище, ею обволакивают яйца, образуя яйцевой кокон. Маленькие паучки выделяют тонкую паутинку в виде нити, с помощью которой они разносятся ветром на далекие расстояния. Паутину используется для ловли добычи, с помощью паутинных гамаков самцы заполняют спермой свои семенные капсулы (резервуары) на педипальпах.

Паутинные железы выделяют клейкое тянущееся вещество, которое затвердевает на воздухе. У пауков имеется несколько типов паутинных

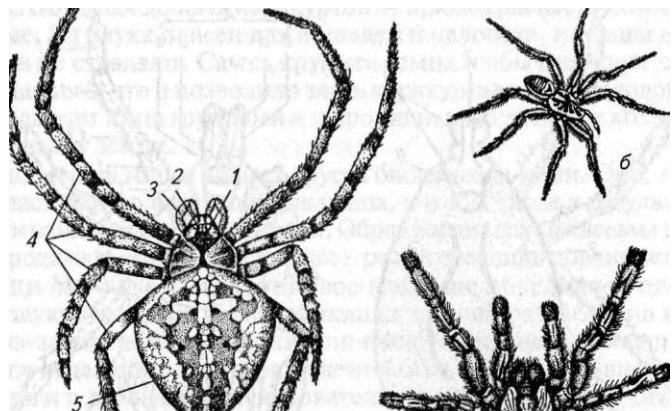


Рис. 109. Представители отряда Пауки:
 а — паук-крестовик *Araneus diadematus*; 1 — головогрудь; 2 — хелицеры; 3 — педипальпы;
 4 — ходильные ноги; 5 — брюшко; б — тарантул *Lycosa singoriensis*; в — каракурт (самка в
 гнезде у яйцевых коконов); г — паук-птицеед *Poecilotheria regalis*

желез разного назначения. Их протоки на бородавках открываются трубочками и конусами. У паука-крестовика на бородавках насчитывают до 560 трубочек и до 20 конусов. Множество выделяемых тончайших паутинок склеиваются в одну общую паутинную нить. Паутинные ииты предназначены для разных целей: изготовления ловчих сетей, постройки яйцевого кокона или жилого дома и т. п. В связи с этим у пауков существует несколько различных типов паутины: сухая, влажная, клейкая, гофрированная, для основания и поперечин ловчих сетей и т. п.

У большинства пауков имеется пара легких и пара трахей (подотряд Двулегочные), у некоторых тропических пауков есть только две пары легочных мешков. В нашей стране обитает около 1,5 тыс. видов пауков,

Рис. 110. Морфология клеша *Hyalomma* (самец) с дорсальной и вентральной сторон:
1 — средняя бороздка; 2 — глаза; 3 — латеральные бороздки; 4, //—лапка; 5 — присоски и коготки; 6 — пальпы; 7 — футляр хелицер; 8 — хоботок; 9 — основание хоботка; 10 — членики пальп (а, б, в, г); 12 — передняя лапка; 13 — голень; 14 — бедро; 15 — вертлуг; 16 — половое отверстие; 17 — половые бороздки; 18 — анальное отверстие

относящихся к двулегочным паукообразным. Наиболее часто встречаются пауки-крестовики (*Araneus*), которых легко узнать по крестообразному рисунку на верхней стороне брюшка (см. рис. 109). Они заселяют леса, кустарники, приусадебные участки и постройки. На ветвях пауки-крестовики сооружают большие радиальные сети, нити которых покрыты клейкой массой, захватывающей попавших в сети насекомых.

Домовые пауки (*Tegenaria*) селятся обычно в жилых и производственных помещениях, натягивая горизонтальные паутинные сети, с помощью которых они ловят мух и других насекомых.

Особую группу образуют пауки, преследующие свою жертву. Обычным представителем на юге страны в степных районах является тарантул (*Lycosa singoriensis*). Живет тарантул в отвесных норках, вырытых в почве и выстланных паутиной. Это самый крупный паук отечественной фауны — до 3 см длиной (см. рис. 109). Укус тарантула вызывает у человека болезненный отек, но не представляет серьезной опасности.

Для человека весьма опасен паук *каракурт* (*Latrodectus tredecimguttatus*), которого можно встретить в степных районах Кавказа и Поволжья. Это средних размеров (около 1,5 см) паук черного цвета с красными пятнышками на верхней поверхности брюшка (рис. 110). Каракуры обитают в норках, а на поверхности почвы расстилают ловчую

паутину. Обычная добыча каракуртов — прямокрылые, в том числе саранчовые. Яд паука опасен для лошадей и человека, но овцы и свиньи от укусов не страдают. Самка крупнее самца и обычно после спаривания съедает его, что и позволило звать каракурта «черной вдовой». Самка откладывает яйца группами в шаровидные паутинные коконы, расположенные у земли.

Пауков очень много во всех ярусах биоценозов суши. Они, как хищники, участвуют во всех пищевых цепях, и в том числе в регуляции численности самых разных насекомых. Образ жизни пауков весьма разнообразен: бродяжки, сидячие (тенетные), растягивающие ловчие сети и др.

КЛЕЩИ (ACARINA). Это групповое название объединяет три отряда класса паукообразных. Классификация клещей разработана недостаточно из-за скудности сведений по филогении, неравномерной изученности отдельных групп, появления большого числа вновь открытых видов и т. д. Многие исследователи не без оснований считают, что в классе Паукообразные следует выделить два подкласса: Пауки (Агапеа) и Клещи (Acari). От других подклассов клещи отличаются слабой выраженностью и полным отсутствием сегментации брюшного отдела тела. Постэмбриональное развитие клещей в отличие от других паукообразных проходит с метаморфозом: личинка имеет три пары ног.

Общая характеристика клещей. По приспособленности к различным местам обитания клещи приближаются к насекомым. Они живут во мхах и в лишайниках, лесной подстилке, в почвенном слое они могут составлять 95 % фауны членистоногих. Клещи живут во всех водоемах, даже в горячих источниках вулканов. Заселили они многие органы и ткани растений, разнообразны их отношения с растениями и животными. Многие растительноядные клещи наносят серьезный вред сельскохозяйственному производству. Потери урожая достигают 20—30 % в тепличных хозяйствах из-за паутинного клеша, вредящего овощным культурам. Повреждения, наносимые грушевым галловым клещом, приводят к потери до 90 % урожая. Земляничный клещ снижает сбор ягод на 40—70 %. Большой ущерб наносит зернопродуктам группа амбарных клещей. Часть клещей является паразитами животных и человека.

Покровы клещей, как и покровы всех членистоногих, состоят из кутикулы, гиподермы и подстилающей ее базальной перепонки. Тело некоторых клещей покрыто сверху мощным панциревидным щитом (панцирные клещи). У кровососущих иксодовых клещей покровы слабо склеротизированы, что позволяет самкам сильно растягиваться во время сосания крови. К производным наружных покровов относятся щетинки, щитки и железы.

Нервная система. Характерной особенностью клещей является то, что их центральная нервная система представляет собой единую цельную массу нервной ткани, окружающую пищевод плотным кольцом — **мозгом**. У клещей обнаружены чувствующие органы, связанные с механическим, химическим, гигротермическим чувствами и зрением. Основу органов чувств составляют сенсиллы, состоящие из щетинок,

пор и т. д. У клещей простые глаза, как правило, их две пары. Представители большого числа видов лишены глаз.

Пищеварительная система. Передняя кишечка подразделяется на мускулистую глотку и пищевод, который впадает в среднюю кишечку. У части клещей пищевод образует расширение — зоб. Средняя кишечка, часто называемая желудком, у многих видов связана со слепыми отростками, которые особенно хорошо развиты у кровососущих клещей. Их насчитываются до семи пар, и они значительно превышают размером саму среднюю кишечку. Небольшие отростки имеют сапрофаги.

Сложно устроена средняя кишечка у питающихся соком растений клещей, обладающих системой разделения пищи на фракции. Задний отдел кишечника состоит из тонкой и толстой кишечек, между которыми впадают в кишечник мальпигиевые сосуды. Очень короткая прямая кишечка выстлана хитином и заканчивается анальным отверстием. Для отдельных клещей характерно внекишечное пищеварение.

Ротовые части представлены хелицерами и педипальпами и могут образовывать два типа ротовых аппаратов: грызущий и колюще-сосущий. Ротовой аппарат грызущего типа характерен для клещей, питающихся твердой растительной пищей. С переходом к питанию жидккой пищей ротовые части становятся тонкими и теряют зубцы, образуя стилеты. Парные стилеты в виде двух желобков составляют трубку, которая выдвигается вперед и погружается в ткань растения.

Строение ног. Большинство клещей имеют четыре пары ног, а их личинки — три пары ног. Встречаются виды с редукцией части ног. Ноги состоят из шести члеников: тазика, вертлуга, бедра, колена, голени, лапки. Отдельные членики иногда сливаются с образованием четырех- или пятичлениковой конечности. Лапка может иметь различные приспособления для передвижения: пару коготков (у многих видов), присоски (у эмподий), расположенные между коготками, липкий секрет для удержания на субстрате, паутинные выделения и т. д. У водных клещей ноги превращаются в плавательные. Могут быть задние — прыгательные, могут быть приспособления для удержания на волосах, перьях и т. п. (см. рис. 110).

Кровеносная система лакунного типа. У большинства видов отсутствуют не только сосуды, но и сердце. Кровь бесцветная.

Органы дыхания. Крупные клещи дышат с помощью трахей. Мелкие формы дышат через покровы тела. Дыхальца — стигмы — парные, всего их от одной до четырех пар.

Органы выделения — мальпигиевые сосуды и коксальные железы.

Органы размножения. Все клещи раздельнополы. Часто наблюдается половой диморфизм по многим признакам, в том числе и по размерам тела. Половое отверстие обычно находится на уровне задней пары ног на брюшной стороне тела. У самки и самца оно зачастую прикрыто специальными щитками или клапанами, покрытыми сенсорными щетинками. Сперматофоры вводятся в половое отверстие самок с помощью хелицер самца. У некоторых клещей самцы прикрепляют сперматофоры к субстрату. У панцирных клещей имеется длинный яйцеклад.

Большинство клещей откладывают яйца, и только у немногих отмечено живорождение. Для ряда панцирных клещей характерно посмертное живорождение. В этом случае самка погибает, не отложив яиц, и они развиваются внутри ее тела. Вылупившиеся личинки, защищенные от неблагоприятных условий панцирем матери, питаются ее тканями, а затем выходят наружу. У клещей встречается партеногенез.

В течение онтогенеза клещи проходят четыре фазы: яйца, личинки, нимфы и взрослого клеша (имаго). По окончании эмбриогенеза личинка выходит из яйца. У некоторых видов еще в яйце происходит эмбриональная линька зародыша. Вышедшая личинка отличается от взрослого клеша меньшим числом щетинок, меньшими размерами, отсутствием полового отверстия и последней пары ног. Личинки хищных и паразитических видов не питаются и живут за счет запасов желтка. Закончившая развитие личинка впадает в состояние покоя, линяет и превращается в нимфу.

Нимфа имеет четыре пары ног, на ее теле появляются зачатки гениталий, увеличивается число щетинок. Фаза нимфы может включать от одного до трех возрастов: нимфа первого (протонимфа), нимфа второго (дейтонимфа) и третьего (тритонимфа) возраста. Переход из одного возраста в другой сопровождается периодом покоя и линькой.

При наступлении неблагоприятных условий нимфа первого возраста переходит в особую стадию *гипопуса*. Гипопусы значительно отличаются от обычных нимфальных фаз клещей: они лишены действующего ротового аппарата, у них сильно редуцируется пищеварительный аппарат и существуют они за счет резервов тела. Гипопусы бывают подвижные (расселительные) и покоящиеся (рис. 111).

Подвижные гипопусы сохраняют развитые ноги, покровы их уплотняются, тело уплощается. Передвигаются активно и пассивно, прикрепляясь к различным животным с помощью присосок или других приспособлений.

Для покоящихся гипопусов характерна сильная редукция ног (мучной клеш) или их полная утрата (волосатый домовый клеш). В состоянии покоя они могут находиться месяцами и устойчивы даже к действию фунгицидов. В благоприятных условиях гипопус линяет и превращается в нимфу, которая после стадии покоя и линьки превращается во взрослого клеша.

У многих клещей, у которых хорошо развит половой диморфизм, самцы развиваются быстрее самок. Поэтому к моменту последней линьки самки самцы уже достигают половой зрелости. Вскоре после сбрасывания самкой личиночной шкурки происходит спаривание, а через 2—3 сут самки начинают откладку яиц.

Жизненные циклы клещей разнообразны, зависят от многих факторов. Есть виды паутинных клещей, которые в течение года могут дать до 20 поколений. В северных ареалах на развитие одной генерации иксодовых клещей требуется до 4 лет.

В жизненном цикле клещей могут быть неблагоприятные периоды. Их это случае все процессы жизнедеятельности клещей резко замедля-

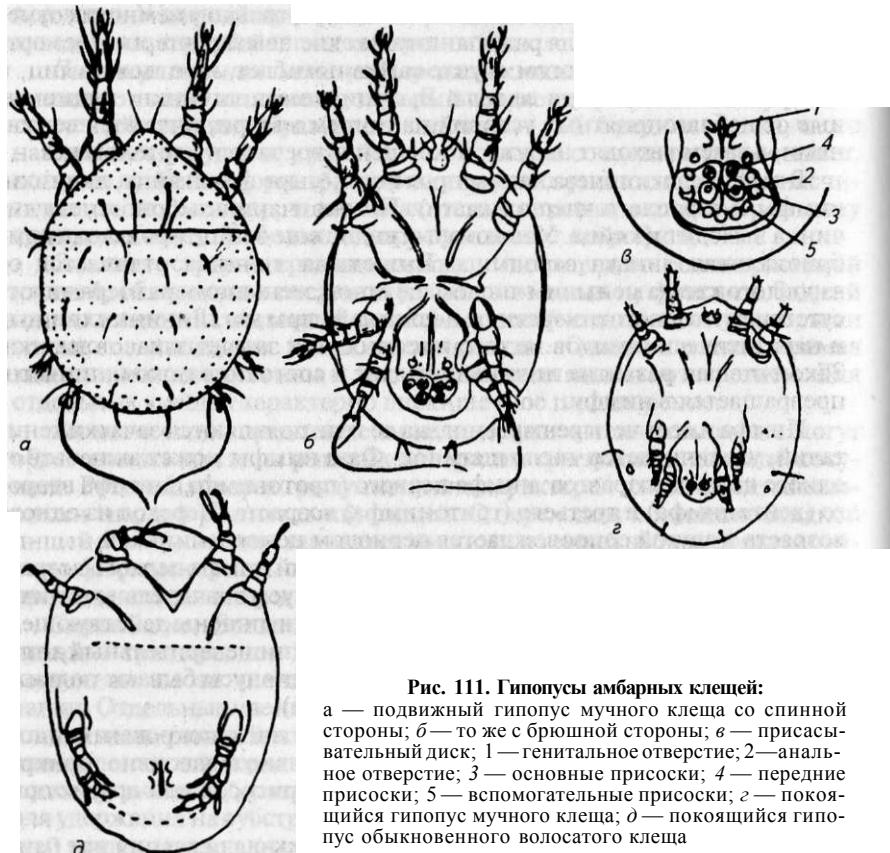


Рис. 111. Гипопусы амбарных клещей:
 а — подвижный гипопус мучного клеща со спинной стороны; б — то же с брюшной стороны; в — присасывательный диск; 1 — генитальное отверстие; 2 — анальное отверстие; 3 — основные присоски; 4 — передние присоски; 5 — вспомогательные присоски; г — покоящийся гипопус мучного клеща; д — покоящийся гипопус обыкновенного волосатого клеща

ются и они переходят в состояние диапаузы. В отличие от обычного покоя (оцепенения) диапауза находится в сложных взаимоотношениях с внешней средой, так как именно эти взаимоотношения обеспечивают синхронность развития организма клещей с фенологией питающих растений и климатическими условиями среды. Диапауза может быть зимней и летней.

Некоторые виды клещей ведут хищнический образ жизни, питаясь другими клещами и мелкими насекомыми. Поражая клещей-вредителей и паразитов сельскохозяйственных растений, хищные формы клещей приносят существенную пользу и могут быть использованы в биологической борьбе с вредными членистоногими. Многие почвенные клещи, являясь сапрофагами, участвуют в почвообразовательных процессах.

Отряд Клещи-сенокосцы (Opiliocarina) включает небольшую группу примитивных клещей с сохранившейся сегментацией тела: два последних сегмента головогруди свободные и брюшко из восьми сегментов. У

них две пары глаз, четыре пары стигм, хелицеры клешневидные, а между ними расположен теркообразный орган. Первая пара ног специализирована в основном как сенсорные органы. Размеры тела сенокосцев достигают 1 мм. Живут скрытно, обитая под камнями, в почвенном слое. Их хозяйственное значение неопределено.

Отряд Акариформные клещи (Acariformes). Это наиболее крупный отряд, объединяющий разнообразные по морфологическим и экологическим особенностям формы и насчитывающий более 15 тыс. видов. Наряду с микроскопическими мелкими паразитическими видами (менее 0,1 мм длины) встречаются свободноживущие хищники, достигающие в длину 10 мм. Примитивные формы дышат через покровы тела, а у эволюционно прогрессивных форм дыхание осуществляется с помощью трахей. Размножение сперматофорами. Развитие с анаморфозом. В состав отряда входят два подотряда: Краснотелковые (*Trombidiformes*) и Саркоптоидные (*Sarcoptiformes*), включающие множество семейств.

Подотряд Краснотелковые (*Trombidiformes*) включает много семейств свободноживущих сухопутных и водных клещей, имеются среди них и паразиты. Хелицеры колющие. Одна пара дыхальца расположена в передней части тела.

Из растительноядных клещей существенный вред растениям причиняют паутинные клещи из сем. *Tetranychidae*. Это мелкие клещики (0,3—0,5 мм) разнообразной окраски. Большинство видов выделяют паутину, что и дало название семейству. Под слоем паутины на нижней стороне листьев растений клещи образуют колонии. Из яиц выходят личинки, которые проходят стадии нимфы первого и второго возрастов и затем становятся взрослыми клещами. Весь цикл развития длится 12—28 сут. Клещи питаются соком растений, который высасывают из листьев. Особенно большой вред приносят в тепличных и парниковых хозяйствах (рис. 112). Серьезными вредителями являются обыкновенный паутинный (*Tetranychus urticae* Koch), садовый паутинный (*Schizotetranychus* Ond.) и другие.

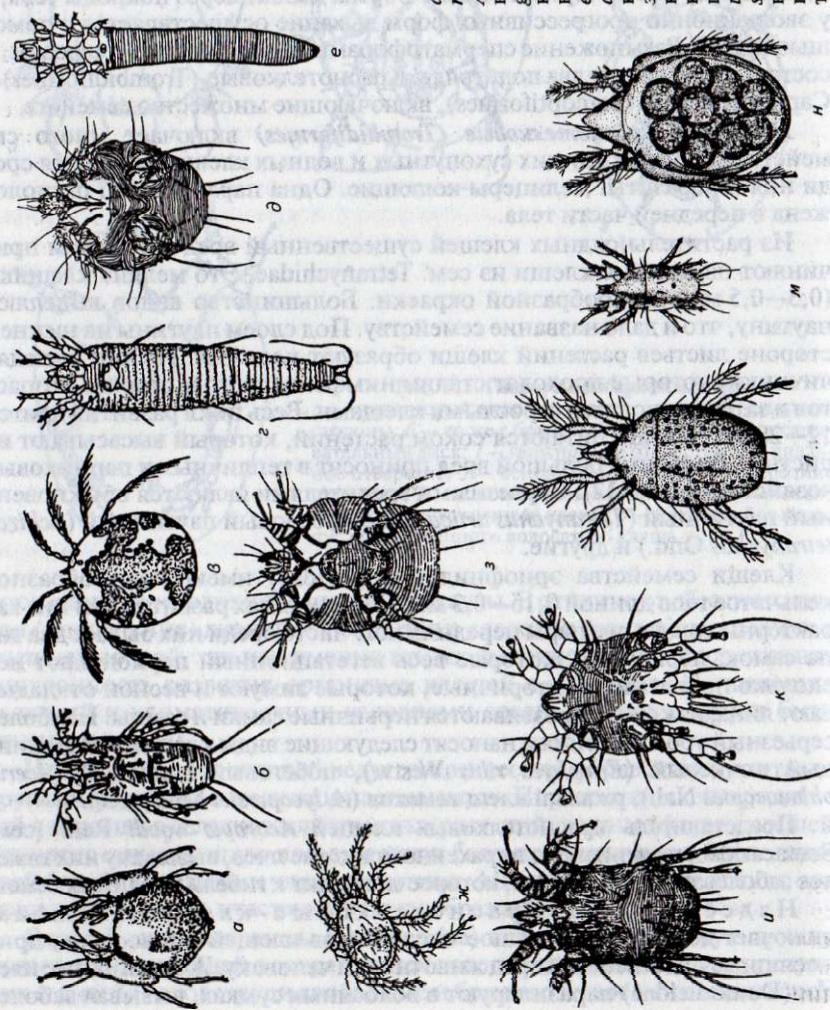
Клещи семейства эриофиид (*Eriophyidae*) имеют червеобразное кольчатое тело длиной 0,15—0,3 мм. На всех фазах развития для них характерны только две пары передних ног. Часто среди них бывает два типа самок: первичные, которые весь вегетационный период дают несколько поколений, и вторичные, которые зимуют и весной откладывают яйца, из которых развиваются первичные самки и самцы. Наиболее серьезный урон растениям наносят следующие виды клещей: смородинный почковый (*Eriophyes ribis* Westw.), побеговый слиновый (*Aceria phloeocoptes* Nal.), ржавый клещ томатов (*A. lycopersici* Mas.) и др.

Представитель краснотелковых клещей *Acarapis woodi* Ren. (сем. *Scutacaridae*) паразитирует в трахейной системе пчел, вызывая у них тяжелое заболевание акарапидоз, которое приводит к гибели пчелиных семей.

Надсемейство Краснотелковые клещи (Trombea) включает достаточно большое число видов клещей-кровососов, приносящих существенный вред животным и человеку. Железничные клещи (*Demodicidae*) паразитируют в волосяных сумках, вызывая заболе-

Рис. 112. Клещи:

a — панцирный клещ *Galumna muscicola*; *b* — первоый клещ *Aneloposis passerinus*; *c* — волчаной клещ *Hydrachna geographica*; *d* — четырехногий клещ *Eriophyes*; *e* — чесоточный зудень *Sarcopis scadiei*; *f* — железнина угря *Demodex felicis*; *g* — группный клещ *Poecilococcus necrotropicus*; *h* — зудневый клещ; *i* — на кожниковый клещ; *k* — клещ коксед; *l* — обыкновенный паутинный клещ *Tetranychus cinnabarinus*; *m* — мучной клещ *Acarus siro*; *n* — панцирный (орбатидный) клещ, внутри которого находится цистицеркоиды ленточного черва мониезии



вание — железницу у животных и человека (см. рис. 112). При сильном поражении на 1 см² кожи овец насчитывали до 25 тыс. особей клещей. В пораженных местах шерсть выпадает, в дерме кожи образуются пузырьки.

Подотряд Саркоптоидные (Sarcoptiformes) включает клещей, имеющих грызущий ротовой аппарат. Среди большого числа семейств следует выделить панцирных клещей (*Oribatei*) — участников почвообразовательных процессов, среди которых есть виды, являющиеся промежуточными хозяевами паразитических ленточных червей.

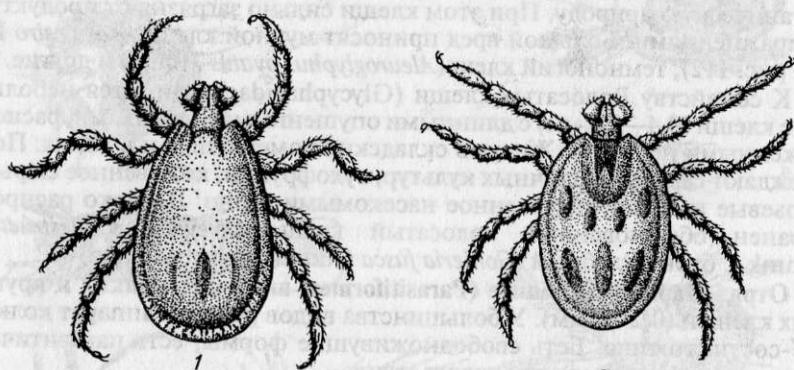
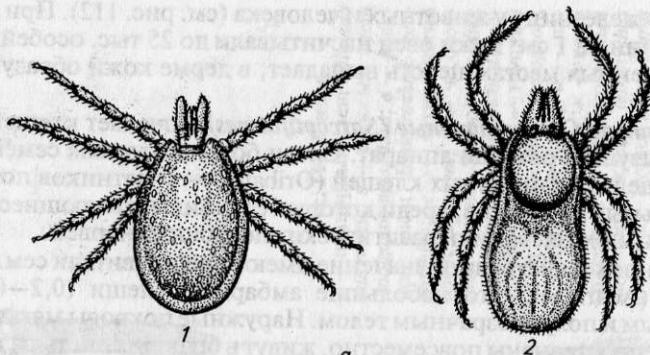
Важное хозяйственное значение имеют представители сем. Мучные клещи (Acaridae). Это небольшие амбарные клещи (0,2—0,8 мм) с овальным и полупрозрачным телом. Наружные покровы мягкие и гладкие. Распространены повсеместно, живут в опавших листьях, в скирдах сена и соломы, в зернохранилищах, складах, гнездах птиц и т. д., где повреждают зернофураж, муку, сухофрукты и многое другое, имеющее органическую природу. При этом клещи сильно загрязняют продукты испражнениями. Большой вред приносят мучной клещ (*Acarus siro* L, см. рис. 112), темноногий клещ (*Aleuroglyphus ovatus* Troup.) и другие.

К семейству Волосатые клещи (Glycyphagidae) относятся небольшие клещи (0,4—0,6 мм) с длинными опущенными щетинками, расположеннымми на спине. Живут в складских помещениях и в домах. Повреждают семена масличных культур, сухофрукты, кожевенное сырье, перьевые изделия, пораженное насекомыми зерно. Широко распространен обыкновенный волосатый клещ (*Glycyphagus destructor* Sehrnk.), бурый хлебный (*Gohieria fusca* Oud.) и другие клещи.

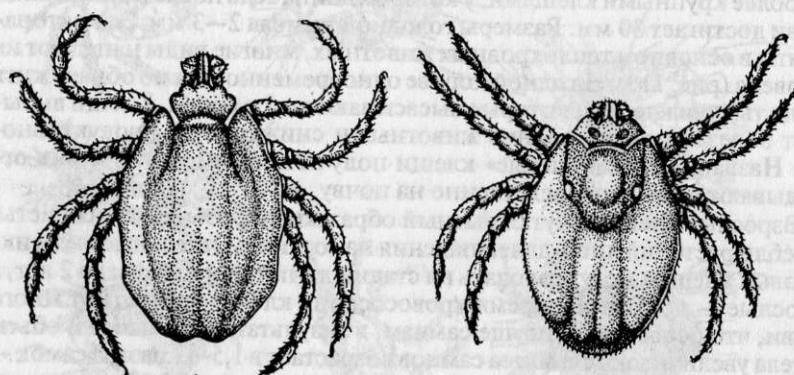
Отряд Паразитiformные (Parasitiformes) включает мелких и крупных клещей (0,2—7 мм). У большинства видов ротовой аппарат колюще-сосущего типа. Есть свободноживущие формы, есть паразитические виды и вредители продовольствия.

Семейство Иксодовые, или Пастбищные клещи (Ixodidae), относящиеся к подотряду Заднедыхальцевые (Metastigmata), представлено наиболее крупными клещами, у которых длина тела после насасывания крови достигает 30 мм. Размеры голодной нимфи 2—3 мм. Это эктопаразиты в основном теплокровных животных, многие виды нападают на человека (рис. 113). На одной корове одновременно можно обнаружить около тысячи клещей, которые высасывают до 5 л крови. Клещи вызывают болезненное состояние животных и снижение их продуктивности. Название «пастбищные» клещи получили потому, что самки откладывают яйца непосредственно на почву

Взрослые клещи ведут скрытный образ жизни, взбираясь на листья и стебли растений лишь для нападения на хозяина. Отдельные виды иксодовых клещей могут голодать на стадии личинки и нимфи до 2 лет, а взрослые — до года. Во время кровососания клещи поглощают много крови, что особенно присуще самкам, в результате чего масса и объем их тела увеличиваются: масса самцов возрастает в 1,5—2 раза, а самок — в 100 раз. Большинство видов в период развития и во взрослом состоянии имеют разных хозяев. По биологическому признаку различают од-



6



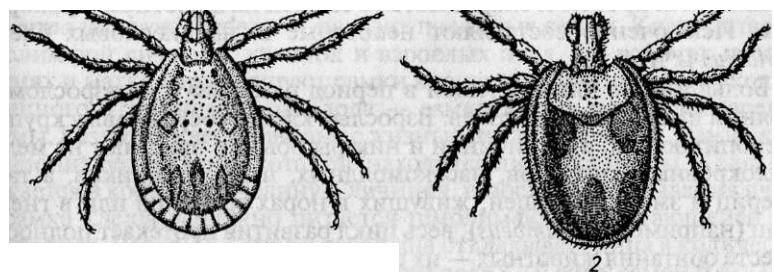


Рис. 113. Представители клещей подотряда Ixodidae:
a — Ixodes ricinus; б — Haemaphysalis punctata; в — Boophilus calcaratus; г — Dermacentor dagestanicus; д — Rhipicephalus tyrranicus. 1 — самец; 2 — самка

но-, двух- и треххозяиных клещей. Однохозяиные клещи все стадии своего развития проходят на одном хозяине. Так, личинки *Boophilus calcaratus* нападают на крупных животных и после насыщения кровью линяют, оставаясь на этом же хозяине, и превращаются в нимфу. Нимфы питаются и превращаются на этом же хозяине в имаго. Лишь в фазе сытых половозрелых особей клещи самопроизвольно отпадают со своего единственного хозяина и откладывают в почве яйца.

У двуххозяиных клещей на теплокровное животное нападают личинки, а с него отпадают уже сытые нимфы. Нимфы в почве линяют, превращаются в имаго, которые нападают на второго хозяина и после насыщения кровью становятся половозрелыми клещами.

Треххозяиный тип питания (большинство *Ixodes*) заключается в том, что клещи нападают на животных в каждой фазе развития, каждый раз отпадая и линяя во внешней среде. Половозрелый клещ должен насосаться крови, так как лишь после этого у него созревают половые продукты. Насасывание кровью сопровождается у самок быстрым развитием яичников, что позволяет обеспечивать высокую плодовитость. После этого клещи спариваются и самки откладывают яйца, число ко-

ЧАСТЬ I

МАТЕРИАЛЫ ДЛЯ ОБЯЗАТЕЛЬНОГО ЧТЕНИЯ



Обработка ошибок

Прежде чем изучать функции, предлагаемые Microsoft Windows, посмотрим, как в них устроена обработка ошибок.

Когда Вы вызываете функцию Windows, она проверяет переданные ей параметры, а затем пытается выполнить свою работу. Если Вы передали недопустимый параметр или если данную операцию нельзя выполнить по какой-то другой причине, она возвращает значение, свидетельствующее об ошибке. В таблице 1-1 показаны типы данных для возвращаемых значений большинства функций Windows.

Тип данных	Значение, свидетельствующее об ошибке
VOID	Функция всегда (или почти всегда) выполняется успешно. Таких функций в Windows очень мало.
BOOL	Если вызов функции заканчивается неудачно, возвращается 0; в остальных случаях возвращаемое значение отлично от 0. (Не пытайтесь проверять его на соответствие TRUE или FALSE.)
HANDLE	Если вызов функции заканчивается неудачно, то обычно возвращается NULL; в остальных случаях HANDLE идентифицирует объект, которым Вы можете манипулировать. Будьте осторожны: некоторые функции возвращают HANDLE со значением INVALID_HANDLE_VALUE, равным –1. В документации Platform SDK для каждой функции четко указывается, что именно она возвращает при ошибке –NULL или INVALID_HANDLE_VALUE.
PVOID	Если вызов функции заканчивается неудачно, возвращается NULL; в остальных случаях PVOID сообщает адрес блока данных в памяти.
LONG или DWORD	Это значение — «крепкий орешек». Функции, которые сообщают значения каких-либо счетчиков, обычно возвращают LONG или DWORD. Если по какой-то причине функция не сумела сосчитать то, что Вы хотели, она обычно возвращает 0 или –1 (все зависит от конкретной функции). Если Вы используете одну из таких функций, проверьте по документации Platform SDK, каким именно значением она уведомляет об ошибке.

Таблица 1-1. Стандартные типы значений, возвращаемых функциями Windows

При возникновении ошибки Вы должны разобраться, почему вызов данной функции оказался неудачен. За каждой ошибкой закреплен свой код — 32-битное число.

Функция Windows, обнаружив ошибку, через механизм локальной памяти потока сопоставляет соответствующий код ошибки с вызывающим потоком. (Локальная память потока рассматривается в главе 21.) Это позволяет потокам работать независимо друг от друга, не вмешиваясь в чужие ошибки. Когда функция вернет Вам управление, ее возвращаемое значение будет указывать на то, что произошла какая-то ошибка. Какая именно — Вы узнаете, вызвав функцию *GetLastError*:

```
DWORD GetLastError();
```

Она просто возвращает 32-битный код ошибки для данного потока.

Теперь, когда у Вас есть код ошибки, Вам нужно обменять его на что-нибудь более приятное. Список кодов ошибок, определенных Microsoft, содержится в заголовочном файле WinError.h. Я приведу здесь его небольшую часть, чтобы Вы представляли, на что он похож.

```
// MessageId: ERROR_SUCCESS
//
// MessageText:
//
// The operation completed successfully.
//
#define ERROR_SUCCESS           0L

#define NO_ERROR                // dderror
#define SEC_E_OK                 ((HRESULT)0x00000000L)

//
// MessageId: ERROR_INVALID_FUNCTION
//
// MessageText:
//
// Incorrect function.
//
#define ERROR_INVALID_FUNCTION   1L    // dderror

//
// MessageId: ERROR_FILE_NOT_FOUND
//
// MessageText:
//
// The system cannot find the file specified.
//
#define ERROR_FILE_NOT_FOUND     2L

//
// MessageId: ERROR_PATH_NOT_FOUND
//
// MessageText:
//
// The system cannot find the path specified.
//
#define ERROR_PATH_NOT_FOUND     3L

//
// MessageId: ERROR_TOO_MANY_OPEN_FILES
//
// MessageText:
//
// The system cannot open the file.

//
#define ERROR_TOO_MANY_OPEN_FILES 4L
```

см. след. стр.

```
//  
// MessageId: ERROR_ACCESS_DENIED  
//  
// MessageText:  
//  
// Access is denied.  
//  
#define ERROR_ACCESS_DENIED 5L
```

Как видите, с каждой ошибкой связаны идентификатор сообщения (его можно использовать в исходном коде для сравнения со значением, возвращаемым *GetLastError*), текст сообщения (описание ошибки на нормальном языке) и номер (вместо него лучше использовать индентификатор). Учтите, что я показал лишь крошечную часть файла WinError.h; на самом деле в нем более 21 000 строк!

Функцию *GetLastError* нужно вызывать сразу же после неудачного вызова функции Windows, иначе код ошибки может быть потерян.



GetLastError возвращает последнюю ошибку, возникшую в потоке. Если этот поток вызывает другую функцию Windows и все проходит успешно, код последней ошибки не перезаписывается и не используется как индикатор благополучного вызова функции. Лишь несколько функций Windows нарушают это правило и все же изменяют код последней ошибки. Однако в документации Platform SDK утверждается обратное: якобы после успешного выполнения API-функции обычно изменяют код последней ошибки.

**WINDOWS
98**

Многие функции Windows 98 на самом деле реализованы в 16-разрядном коде, унаследованном от операционной системы Windows 3.1. В нем не было механизма, сообщающего об ошибках через некую функцию наподобие *GetLastError*, и Microsoft не стала «исправлять» 16-разрядный код в Windows 98 для поддержки обработки ошибок. На практике это означает, что многие Win32-функции в Windows 98 не устанавливают код последней ошибки после неудачного завершения, а просто возвращают значение, которое свидетельствует об ошибке. Поэтому Вам не удастся определить причину ошибки.

Некоторые функции Windows всегда завершаются успешно, но по разным причинам. Например, попытка создать объект ядра «событие» с определенным именем может быть успешна либо потому, что Вы действительно создали его, либо потому, что такой объект уже есть. Но иногда нужно знать причину успеха. Для возврата этой информации Microsoft предпочла использовать механизм установки кода последней ошибки. Так что и при успешном выполнении некоторых функций Вы можете вызывать *GetLastError* и получать дополнительную информацию. К числу таких функций относится, например, *CreateEvent*. О других функциях см. Platform SDK.

На мой взгляд, особенно полезно отслеживать код последней ошибки в процессе отладки. Кстати, отладчик в Microsoft Visual Studio 6.0 позволяет настраивать окно Watch так, чтобы оно всегда показывало код и описание последней ошибки в текущем потоке. Для этого надо выбрать какую-нибудь строку в окне Watch и ввести «@err,hr». Теперь посмотрите на рис. 1-1. Видите, я вызвал функцию *CreateFile*. Она вернула значение INVALID_HANDLE_VALUE (-1) типа HANDLE, свидетельствующее о том, что ей не удалось открыть заданный файл. Но окно Watch показывает нам код последней ошибки (который вернула бы функция *GetLastError*, если бы я ее вызвал),

равный 0x00000002, и описание «The system cannot find the file specified» («Система не может найти указанный файл»). Именно эта строка и определена в заголовочном файле WinError.h для ошибки с кодом 2.

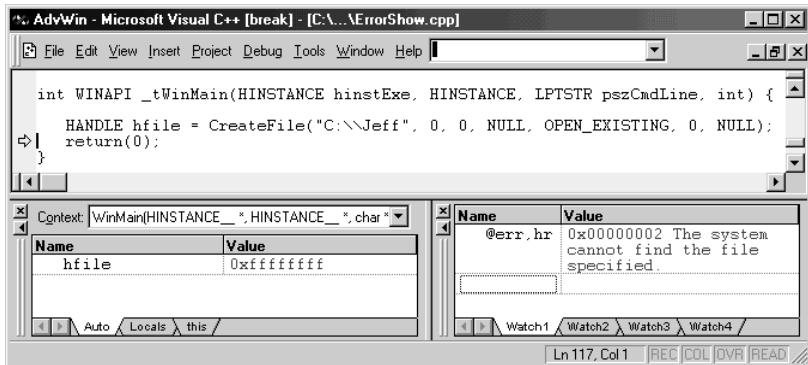
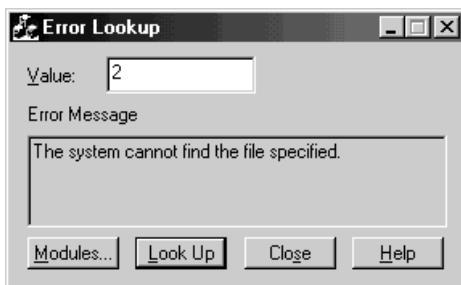


Рис. 1-1. Используя «@err,hr» в окне Watch среды Visual Studio 6.0, Вы можете просматривать код последней ошибки в текущем потоке

С Visual Studio поставляется небольшая утилита Error Lookup, которая позволяет получать описание ошибки по ее коду.



Если приложение обнаруживает какую-нибудь ошибку, то, как правило, сообщает о ней пользователю, выводя на экран ее описание. В Windows для этого есть специальная функция, которая «конвертирует» код ошибки в ее описание, — *FormatMessage*.

```
DWORD FormatMessage(
    DWORD dwFlags,
    LPCVOID pSource,
    DWORD dwMessageId,
    DWORD dwLanguageId,
    PTSTR pszBuffer,
    DWORD nSize,
    va_list *Arguments);
```

FormatMessage — весьма богатая по своим возможностям функция, и именно ее желательно применять при формировании всех строк, показываемых пользователю. Дело в том, что она позволяет легко работать со множеством языков. *FormatMessage* определяет, какой язык выбран в системе в качестве основного (этот параметр задается через апллет Regional Settings в Control Panel), и возвращает текст на соответствующем языке. Разумеется, сначала Вы должны перевести строки на нужные языки и встроить этот ресурс в свой EXE- или DLL-модуль, зато потом функция будет автоматически выбирать требуемый язык. Программа-пример ErrorShow, приведенная в кон-

це главы, демонстрирует, как вызывать эту функцию для получения текстового описания ошибки по ее коду, определенному Microsoft.

Время от времени меня кто-нибудь да спрашивает, составит ли Microsoft полный список кодов всех ошибок, возможных в каждой функции Windows. Ответ: увы, нет. Скажу больше, такого списка никогда не будет — слишком уж сложно его составлять и поддерживать для все новых и новых версий системы.

Проблема с подобным списком еще и в том, что Вы вызываете одну API-функцию, а она может обратиться к другой, та — к третьей и т. д. Любая из этих функций может завершиться неудачно (и по самым разным причинам). Иногда функция более высокого уровня сама справляется с ошибкой в одной из вызванных ею функций и в конечном счете выполняет то, что Вы от нее хотели. В общем, для создания такого списка Microsoft пришлось бы проследить цепочки вызовов в каждой функции, что очень трудно. А с появлением новой версии системы эти цепочки нужно было бы пересматривать заново.

Вы тоже можете это сделать

О'кэй, я показал, как функции Windows сообщают об ошибках. Microsoft позволяет Вам использовать этот механизм и в собственных функциях. Допустим, Вы пишете функцию, к которой будут обращаться другие программы. Вызов этой функции может по какой-либо причине завершиться неудачно, и Вам тоже нужно сообщать об ошибках. С этой целью Вы просто устанавливаете код последней ошибки в потоке и возвращаete значение FALSE, INVALID_HANDLE_VALUE, NULL или что-то другое, более подходящее в Вашем случае. Чтобы установить код последней ошибки в потоке, Вы вызываете *SetLastError*:

```
VOID SetLastError(DWORD dwErrCode);
```

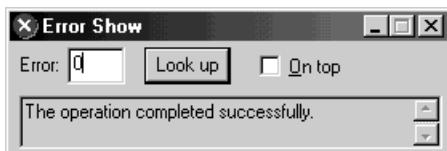
и передаете ей нужное 32-битное число. Я стараюсь использовать коды, уже определенные в WinError.h, — при условии, что они подходят под те ошибки, о которых могут сообщать мои функции. Если Вы считаете, что ни один из кодов в WinError.h не годится для ошибки, возможной в Вашей функции, определите свой код. Он представляет собой 32-битное значение, которое разбито на поля, показанные в следующей таблице.

Биты	31–30	29	28	27–16	15–0
Содержимое:	Код степени «тяжести» (severity)	Кем определен — Microsoft или пользователем	Зарезервирован	Код подсистемы (facility code)	Код исключения
Значение:	0 = успех 1 = информация 2 = предупреждение 3 = ошибка	0 = Microsoft 1 = пользователь	Должен быть 0	Определяется Microsoft	Определяется Microsoft или пользователем

Подробнее об этих полях я рассказываю в главе 24. На данный момент единственное важное для Вас поле — бит 29. Microsoft обещает, что все коды ошибок, генерируемые ее функциями, будут содержать 0 в этом бите. Если Вы определяете собственный код ошибки, запишите сюда 1. Тогда у Вас будет гарантия, что Ваш код ошибки не войдет в конфликт с кодом, определенным Microsoft, — ни сейчас, ни в будущем.

Программа-пример ErrorShow

Эта программа, «01 ErrorShow.exe» (см. листинг на рис. 1-2), демонстрирует, как получить текстовое описание ошибки по ее коду. Файлы исходного кода и ресурсов программы находятся в каталоге 01-ErrorShow на компакт-диске, прилагаемом к книге. Программа ErrorShow в основном предназначена для того, чтобы Вы увидели, как работают окно Watch отладчика и утилита Error Lookup. После запуска ErrorShow открывается следующее окно.



В поле Error можно ввести любой код ошибки. Когда Вы щелкнете кнопку Look Up, внизу, в прокручиваемом окне появится текст с описанием данной ошибки. Единственная интересная особенность программы заключается в том, как она обращается к функции *FormatMessage*. Я использую эту функцию так:

```
// получаем код ошибки
DWORD dwError = GetDlgItemInt(hwnd, IDC_ERRORCODE, NULL, FALSE);

HLOCAL hlocal = NULL; // буфер для строки с описанием ошибки

// получаем текстовое описание ошибки
BOOL fOk = FormatMessage(
    FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_ALLOCATE_BUFFER,
    NULL, dwError, MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US),
    (LPTSTR) &hlocal, 0, NULL);

:

if (hlocal != NULL) {
    SetDlgItemText(hwnd, IDC_ERRORTEXT, (PCTSTR) LocalLock(hlocal));
    LocalFree(hlocal);
} else {
    SetDlgItemText(hwnd, IDC_ERRORTEXT, TEXT("Error number not found."));
}
```

Первая строка считывает код ошибки из текстового поля. Далее я создаю экземпляр описателя (handle) блока памяти и инициализирую его значением NULL. Функция *FormatMessage* сама выделяет нужный блок памяти и возвращает нам его описатель.

Вызывая *FormatMessage*, я передаю флаг *FORMAT_MESSAGE_FROM_SYSTEM*. Он сообщает функции, что мне нужна строка, соответствующая коду ошибки, определенному в системе. Кроме того, я передаю флаг *FORMAT_MESSAGE_ALLOCATE_BUFFER*, чтобы функция выделила соответствующий блок памяти для хранения текста. Описатель этого блока будет возвращен в переменной *hlocal*. Третий параметр указывает код интересующей нас ошибки, а четвертый — язык, на котором мы хотим увидеть ее описание.

Если выполнение *FormatMessage* заканчивается успешно, описание ошибки помещается в блок памяти, и я копирую его в прокручиваемое окно, расположенное в нижней части окна программы. А если вызов *FormatMessage* оказывается неудачным,

я пытаюсь найти код сообщения в модуле NetMsg.dll, чтобы выяснить, не связана ли ошибка с сетью. Используя описатель NetMsg.dll, я вновь вызываю *FormatMessage*. Дело в том, что у каждого DLL или EXE-модуля может быть собственный набор кодов ошибок, который включается в модуль с помощью Message Compiler (MC.exe). Как раз это и позволяет делать утилита Error Lookup через свое диалоговое окно Modules.



ErrorShow.cpp

```
/*
Модуль: ErrorShow.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****
```

```
#include "..\CmnHdr.h"      /* см. приложение A */
#include <Windowsx.h>
#include <tchar.h>
#include "Resource.h"

///////////

#define ESM_POKECODEANDLOOKUP    (WM_USER + 100)
const TCHAR g_szAppName[] = TEXT("Error Show");

///////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_ERRORSHOW);

    // не принимаем коды ошибок, состоящие более чем из 5 цифр
    Edit_LimitText(GetDlgItem(hwnd, IDC_ERRORCODE), 5);

    // проверяем, не передан ли код ошибки через командную строку
    SendMessage(hwnd, ESM_POKECODEANDLOOKUP, lParam, 0);
    return(TRUE);
}

///////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    switch (id) {

        case IDCANCEL:
            EndDialog(hwnd, id);
            break;

        case IDC_ALWAYSONTOP:
            SetWindowPos(hwnd, IsDlgButtonChecked(hwnd, IDC_ALWAYSONTOP)
                ? HWND_TOPMOST : HWND_NOTOPMOST, 0, 0, 0, 0, SWP_NOMOVE | SWP_NOSIZE);
            break;
    }
}
```

Рис. 1-2. Программа-пример ErrorShow

Рис. 1-2. продолжение

```

case IDC_ERRORCODE:
    EnableWindow(GetDlgItem(hwnd, IDOK), Edit_GetTextLength(hwndCtl) > 0);
    break;

case IDOK:
    // получаем код ошибки
    DWORD dwError = GetDlgItemInt(hwnd, IDC_ERRORCODE, NULL, FALSE);

    HLOCAL hlocal = NULL;    // буфер для строки с описанием ошибки

    // получаем текстовое описание ошибки
    BOOL fOk = FormatMessage(
        FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_ALLOCATE_BUFFER,
        NULL, dwError, MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US),
        (PTSTR) &hlocal, 0, NULL);

    if (!fOk) {
        // не связана ли ошибка с сетью?
        HMODULE hDll = LoadLibraryEx(TEXT("netmsg.dll"), NULL,
            DONT_RESOLVE_DLL_REFERENCES);

        if (hDll != NULL) {
            FormatMessage(
                FORMAT_MESSAGE_FROM_HMODULE | FORMAT_MESSAGE_FROM_SYSTEM
                | FORMAT_MESSAGE_ALLOCATE_BUFFER, hDll, dwError,
                MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US), (PTSTR) &hlocal, 0, NULL);
            FreeLibrary(hDll);
        }
    }

    if (hlocal != NULL) {
        SetDlgItemText(hwnd, IDC_ERRORTEXT, (PCTSTR) LocalLock(hlocal));
        LocalFree(hlocal);
    } else {
        SetDlgItemText(hwnd, IDC_ERRORTEXT, TEXT("Error number not found."));
    }
    break;
}

///////////////////////////////
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

switch (uMsg) {
    chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
    chHANDLE_DLGMMSG(hwnd, WM_COMMAND, Dlg_OnCommand);

case ESM_POKECODEANDLOOKUP:
    SetDlgItemInt(hwnd, IDC_ERRORCODE, (UINT) wParam, FALSE);
    FORWARD_WM_COMMAND(hwnd, IDOK, GetDlgItem(hwnd, IDOK), BN_CLICKED,

```

см. след. стр.

Рис. 1-2. продолжение

```
        PostMessage();
        SetForegroundWindow(hwnd);
        break;
    }

    return(FALSE);
}

/////////////////////////////// Конец файла ///////////////////////////////
```

```
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    HWND hwnd = FindWindow(TEXT("#32770"), TEXT("Error Show"));
    if (IsWindow(hwnd)) {
        // экземпляр уже выполняется, активизируем его и посыпаем ему новый номер
        SendMessage(hwnd, ESM_POKECODEANDLOOKUP, _ttoi(pszCmdLine), 0);
    } else {
        DialogBoxParam(hinstExe, MAKEINTRESOURCE(IDD_ERRORSHOW),
            NULL, Dlg_Proc, _ttoi(pszCmdLine));
    }
    return(0);
}

/////////////////////////////// Конец файла ///////////////////////////////
```

Unicode

Microsoft Windows становится все популярнее, и нам, разработчикам, надо больше ориентироваться на международные рынки. Раньше считалось нормальным, что локализованные версии программных продуктов выходят спустя полгода после их появления в США. Но расширение поддержки в операционной системе множества самых разных языков упрощает выпуск программ, рассчитанных на международные рынки, и тем самым сокращает задержки с началом их дистрибуции.

В Windows всегда были средства, помогающие разработчикам локализовать свои приложения. Программа получает специфичную для конкретной страны информацию (региональные стандарты), вызывая различные функции Windows, и узнает предпочтения пользователя, анализируя параметры, заданные в Control Panel. Кроме того, Windows поддерживает массу всевозможных шрифтов.

Я решил переместить эту главу в начало книги, потому что вопрос о поддержке Unicode стал одним из основных при разработке любого приложения. Проблемы, связанные с Unicode, обсуждаются почти в каждой главе; все программы-примеры в моей книге «готовы к Unicode». Тот, кто пишет программы для Microsoft Windows 2000 или Microsoft Windows CE, просто обязан использовать Unicode, и точка. Но если Вы разрабатываете приложения для Microsoft Windows 98, у Вас еще есть выбор. В этой главе мы поговорим и о применении Unicode в Windows 98.

Наборы символов

Настоящей проблемой при локализации всегда были операции с различными наборами символов. Годами, кодируя текстовые строки как последовательности однобайтовых символов с нулем в конце, большинство программистов так к этому привыкло, что это стало чуть ли не второй их натурой. Вызываемая нами функция *strlen* возвращает количество символов в заканчивающемся нулем массиве однобайтовых символов. Но существуют такие языки и системы письменности (классический пример — японские иероглифы), в которых столько знаков, что одного байта, позволяющего кодировать не более 256 символов, просто недостаточно. Для поддержки подобных языков были созданы двухбайтовые наборы символов (double-byte character sets, DBCS).

Одно- и двухбайтовые наборы символов

В двухбайтовом наборе символ представляется либо одним, либо двумя байтами. Так, для японской кандзи, если значение первого байта находится между 0x81 и 0x9F или между 0xE0 и 0xFC, надо проверить значение следующего байта в строке, чтобы определить полный символ. Работа с двухбайтовыми наборами символов — просто кошмар для программиста, так как часть их состоит из одного байта, а часть — из двух.

Простой вызов функции *strlen* не дает количества символов в строке — она возвращает только число байтов. В ANSI-библиотеке С нет функций, работающих с двухбайтовыми наборами символов. Но в аналогичную библиотеку Visual C++ включено множество функций (типа *_mbslen*), способных оперировать со строками мультибайтовых (как одно-, так и двухбайтовых) символов.

Для работы с DBCS-строками в Windows предусмотрен целый набор вспомогательных функций:

Функция	Описание
<i>PTSTR CharNext (PCTSTR pszCurrentChar);</i>	Возвращает адрес следующего символа в строке
<i>PTSTR CharPrev (PCTSTR pszStart,</i> <i>PCTSTR pszCurrentChar);</i>	Возвращает адрес предыдущего символа в строке
<i>BOOL IsDBCSLeadByte (BYTE bTestChar);</i>	Возвращает TRUE, если данный байт — первый в DBCS-символе

Функции *CharNext* и *CharPrev* позволяют «перемещаться» по двухбайтовой строке единовременно на 1 символ вперед или назад, а *IsDBCSLeadByte* возвращает TRUE, если переданный ей байт — первый в двухбайтовом символе.

Хотя эти функции несколько облегчают работу с DBCS-строками, необходимость в ином подходе очевидна. Перейдем к Unicode.

Unicode: набор «широких» символов

Unicode — стандарт, первоначально разработанный Apple и Xerox в 1988 г. В 1991 г. был создан консорциум для совершенствования и внедрения Unicode. В него вошли компании Apple, Compaq, Hewlett-Packard, IBM, Microsoft, Oracle, Silicon Graphics, Sybase, Unisys и Xerox. (Полный список компаний — членов консорциума см. на www.Unicode.org.) Эта группа компаний наблюдает за соблюдением стандарта Unicode, описание которого Вы найдете в книге *The Unicode Standard* издательства Addison-Wesley (ее электронный вариант можно получить на том же www.Unicode.org).

Строки в Unicode просты и логичны. Все символы в них представлены 16-битными значениями (по 2 байта на каждый). В них нет особых байтов, указывающих, чем является следующий байт — частью того же символа или новым символом. Это значит, что прохождение по строке реализуется простым увеличением или уменьшением значения указателя. Функции *CharNext*, *CharPrev* и *IsDBCSLeadByte* больше не нужны.

Так как каждый символ — 16-битное число, Unicode позволяет кодировать 65 536 символов, что более чем достаточно для работы с любым языком. Разительное отличие от 256 знаков, доступных в однобайтовом наборе!

В настоящее время кодовые позиции¹ определены для арабского, китайского, греческого, еврейского, латинского (английского) алфавитов, а также для кириллицы (русского), японской каны, корейского хангыль и некоторых других алфавитов. Кроме того, в набор символов включено большое количество знаков препинания, математических и технических символов, стрелок, диакритических и других знаков. Все вместе они занимают около 35 000 кодовых позиций, оставляя простор для будущих расширений.

Эти 65 536 символов разбиты на отдельные группы. Некоторые группы, а также включенные в них символы показаны в таблице.

¹ Кодовая позиция (code point) — позиция знака в наборе символов.

16-битный код	Символы	16-битный код	Символы
0000–007F	ASCII	0300–036F	Общие диакритические
0080–00FF	Символы Latin1	0400–04FF	Кириллица
0100–017F	Европейские латинские	0530–058F	Армянский
0180–01FF	Расширенные латинские	0590–05FF	Еврейский
0250–02AF	Стандартные фонетические	0600–06FF	Арабский
02B0–02FF	Модифицированные литеры	0900–097F	Деванагари

Около 29 000 кодовых позиций пока не заняты, но зарезервированы на будущее. Примерно 6 000 позиций оставлено специально для программистов (на их усмотрение).

Почему Unicode?

Разрабатывая приложение, Вы определенно должны использовать преимущества Unicode. Даже если Вы пока не собираетесь локализовать программный продукт, разработка с прицелом на Unicode упростит эту задачу в будущем. Unicode также позволяет:

- легко обмениваться данными на разных языках;
- распространять единственный двоичный EXE- или DLL-файл, поддерживающий все языки;
- увеличить эффективность приложений (об этом мы поговорим чуть позже).

Windows 2000 и Unicode

Windows 2000 — операционная система, целиком и полностью построенная на Unicode. Все базовые функции для создания окон, вывода текста, операций со строками и т. д. ожидают передачи Unicode-строк. Если какой-то функции Windows передается ANSI-строка, она сначала преобразуется в Unicode и лишь потом передается операционной системе. Если Вы ждете результата функции в виде ANSI-строки, операционная система преобразует строку — перед возвратом в приложение — из Unicode в ANSI. Все эти преобразования протекают скрытно от Вас, но, конечно, на них тратятся и лишнее время, и лишняя память.

Например, функция *CreateWindowEx*, вызываемая с ANSI-строками для имени класса и заголовка окна, должна, выделив дополнительные блоки памяти (в стандартной куче Вашего процесса), преобразовать эти строки в Unicode и, сохранив результат в выделенных блоках памяти, вызвать Unicode-версию *CreateWindowEx*.

Для функций, заполняющих строки выделенные буферы, системе — прежде чем программа сможет их обрабатывать — нужно преобразовать строки из Unicode в ANSI. Из-за этого Ваше приложение потребует больше памяти и будет работать медленнее. Поэтому гораздо эффективнее разрабатывать программу, с самого начала ориентируясь на Unicode.

Windows 98 и Unicode

Windows 98 — не совсем новая операционная система. У нее «16-разрядное наследство», которое не было рассчитано на Unicode. Введение поддержки Unicode в Windows 98 было бы слишком трудоемкой задачей, и при разработке этой операционной системы от нее отказались. По этой причине вся внутренняя обработка строк в Windows 98, как и у ее предшественниц, построена на применении ANSI.

И все же Windows 98 допускает работу с приложениями, обрабатывающими символы и строки в Unicode, хотя вызов функций Windows при этом заметно усложняется. Например, если Вы, обращаясь к *CreateWindowEx*, передаете ей ANSI-строки, вызов проходит очень быстро — не требуется ни выделения буферов, ни преобразования строк. Но для вызова *CreateWindowEx* с Unicode-строками Вам придется самому выделять буфера, явно вызывать функции, преобразующие строки из Unicode в ANSI, обращаться к *CreateWindowEx*, снова вызывать функции, преобразующие строки — на этот раз из ANSI в Unicode, и освобождать временные буфера. Так что в Windows 98 работать с Unicode не столь удобно, как в Windows 2000. Подробнее о преобразованиях строк в Windows 98 я расскажу в конце главы.

Хотя большинство Unicode-функций в Windows 98 ничего не делает, некоторые все же реализованы. Вот они:

- `EnumResourceLanguagesW`
- `EnumResourceNamesW`
- `EnumResourceTypesW`
- `ExtTextOutW`
- `FindResourceW`
- `FindResourceExW`
- `GetCharWidthW`
- `GetCommandLineW`
- `GetTextExtentPoint32W`
- `GetTextExtentPointW`
- `lstrlenW`
- `MessageBoxExW`
- `MessageBoxW`
- `TextOutW`
- `WideCharToMultiByte`
- `MultiByteToWideChar`

К сожалению, многие из этих функций в Windows 98 работают из рук вон плохо. Одни не поддерживают определенные шрифты, другие повреждают область динамически распределяемой памяти (кучу), третьи нарушают работу принтерных драйверов и т. д. С этими функциями Вам придется здорово потрудиться при отладке программы. И даже это еще не значит, что Вы сможете устранить все проблемы.

Windows CE и Unicode

Операционная система Windows CE создана для небольших вычислительных устройств — бездисковых и с малым объемом памяти. Вы вполне могли бы подумать, что Microsoft, раз уж эту систему нужно было сделать предельно компактной, в качестве «родного» набора символов выберет ANSI. Но Microsoft поступила дальновиднее. Зная, что вычислительные устройства с Windows CE будут продаваться по всему миру, там решили сократить затраты на разработку программ, упростив их локализацию. Поэтому Windows CE полностью поддерживает Unicode.

Чтобы не увеличивать ядро Windows CE, Microsoft вообще отказалась от поддержки ANSI-функций Windows. Так что, если Вы пишете для Windows CE, то просто обязаны разбираться в Unicode и использовать его во всех частях своей программы.

В чью пользу счет?

Для тех, кто ведет счет в борьбе Unicode против ANSI, я решил сделать краткий обзор «История Unicode в Microsoft»:

- Windows 2000 поддерживает Unicode и ANSI — Вы можете использовать любой стандарт;
- Windows 98 поддерживает только ANSI — Вы обязаны программировать в расчете на ANSI;

- Windows CE поддерживает только Unicode — Вы обязаны программировать в расчете на Unicode.

Несмотря на то что Microsoft пытается облегчить написание программ, способных работать на всех трех платформах, различия между Unicode и ANSI все равно создают проблемы, и я сам не раз с ними сталкивался. Не поймите меня неправильно, но Microsoft твердо поддерживает Unicode, поэтому я настоятельно рекомендую переходить именно на этот стандарт. Только имейте в виду, что Вас ждут трудности, на преодоление которых потребуется время. Я бы посоветовал применять Unicode и, если Вы работаете в Windows 98, преобразовывать строки в ANSI лишь там, где без этого не обойтись.

Увы, есть еще одна маленькая проблема, о которой Вы должны знать, — COM.

Unicode и COM

Когда Microsoft переносила COM из 16-разрядной Windows на платформу Win32, руководство этой компании решило, что все методы COM-интерфейсов, работающие со строками, должны принимать их только в Unicode. Это было удачное решение, так как COM обычно используется для того, чтобы компоненты могли общаться друг с другом, а Unicode позволяет легко локализовать строки.

Если Вы разрабатываете программу для Windows 2000 или Windows CE и при этом используете COM, то выбора у Вас просто нет. Применяя Unicode во всех частях программы, Вам будет гораздо проще обращаться и к операционной системе, и к COM-объектам.

Если Вы пишете для Windows 98 и тоже используете COM, то попадаете в затруднительное положение. COM требует строк в Unicode, а большинство функций операционной системы — строк в ANSI. Это просто кошмар! Я работал над несколькими такими проектами, и мне приходилось писать прорыв кода только для того, чтобы гонять строки из одного формата в другой.

Как писать программу с использованием Unicode

Microsoft разработала Windows API так, чтобы как можно меньше влиять на Ваш код. В самом деле, у Вас появилась возможность создать единственный файл с исходным кодом, компилируемый как с применением Unicode, так и без него, — достаточно определить два макроса (`UNICODE` и `_UNICODE`), которые отвечают за нужные изменения.

Unicode и библиотека С

Для использования Unicode-строк были введены некоторые новые типы данных. Стандартный заголовочный файл `String.h` модифицирован: в нем определен `wchar_t` — тип данных для Unicode-символа:

```
typedef unsigned short wchar_t;
```

Если Вы хотите, скажем, создать буфер для хранения Unicode-строки длиной до 99 символов с нулевым символом в конце, поставьте оператор:

```
wchar_t szBuffer[100];
```

Он создает массив из ста 16-битных значений. Конечно, стандартные функции библиотеки С для работы со строками вроде `strcpy`, `strchr` и `strcat` оперируют только с ANSI-строками — они не способны корректно обрабатывать Unicode-строки. Поэтому

му в ANSI C имеется дополнительный набор функций. На рис. 2-1 приведен список строковых функций ANSI C и эквивалентных им Unicode-функций.

```
char * strcat(char *, const char *);
wchar_t * wcscat(wchar_t *, const wchar_t *);

char * strchr(const char *, int);
wchar_t * wcschr(const wchar_t *, wchar_t *);

int strcmp(const char *, const char *);
int wcscmp(const wchar_t *, const wchar_t *);

char * strcpy(char *, const char *);
wchar_t * wcscpy(wchar_t *, const wchar_t *);

size_t strlen(const char *);
size_t wcslen(const wchar_t *);
```

Рис. 2-1. Строковые функции ANSI C и их Unicode-аналоги

Обратите внимание, что имена всех новых функций начинаются с *wcs* — это аббревиатура *wide character set* (набор широких символов). Таким образом, имена Unicode-функций образуются простой заменой префикса *str* соответствующих ANSI-функций на *wcs*.



Один очень важный момент, о котором многие забывают, заключается в том, что библиотека C, предоставляемая Microsoft, отвечает стандарту ANSI. А он требует, чтобы библиотека C поддерживала символы и строки в Unicode. Это значит, что Вы можете вызывать функции C для работы с Unicode-символами и строками даже при работе в Windows 98. Иными словами, функции *wcscat*, *wcslen*, *wcstok* и т. д. прекрасно работают и в Windows 98; беспокоиться нужно за функции операционной системы.

Код, содержащий явные вызовы *str*- или *wcs*-функций, просто так компилировать с использованием и ANSI, и Unicode нельзя. Чтобы реализовать возможность компиляции «двойного назначения», замените в своей программе заголовочный файл String.h на TChar.h. Он помогает создавать универсальный исходный код, способный задействовать как ANSI, так и Unicode, — и это единственное, для чего нужен файл TChar.h. Он состоит из макросов, заменяющих явные вызовы *str*- или *wcs*-функций. Если при компиляции исходного кода Вы определяете _UNICODE, макросы ссылаются на *wcs*-функции, а в его отсутствие — на *str*-функции.

Например, в TChar.h есть макрос *_tcscpy*. Если Вы включили этот заголовочный файл, но _UNICODE не определен, *_tcscpy* раскрывается в ANSI-функцию *strcpy*, а если _UNICODE определен — в Unicode-функцию *wcscpy*. В файле TChar.h определены универсальные макросы для всех стандартных строковых функций С. При использовании этих макросов вместо конкретных имен ANSI- или Unicode-функций Ваш код можно будет компилировать в расчете как на Unicode, так и на ANSI.

Но, к сожалению, это еще не все. В файле TChar.h есть дополнительные макросы.

Чтобы объявить символьный массив «универсального назначения» (ANSI/Unicode), применяется тип данных TCHAR. Если _UNICODE определен, TCHAR объявляется так:

```
typedef wchar_t TCHAR;
```

А если _UNICODE не определен, то:

```
typedef char TCHAR;
```

Используя этот тип данных, можно объявить строку символов как:

```
TCHAR szString[100];
```

Можно также определять указатели на строки:

```
TCHAR *szError = "Error";
```

Правда, в этом операторе есть одна проблема. По умолчанию компилятор Microsoft C++ транслирует строки как состоящие из символов ANSI, а не Unicode. В итоге этот оператор normally компилируется, если `_UNICODE` не определен, но в ином случае дает ошибку. Чтобы компилятор сгенерировал Unicode-, а не ANSI-строку, оператор надо переписать так:

```
TCHAR *szError = L"Error";
```

Заглавная буква `L` перед строковым литералом указывает компилятору, что его надо обрабатывать как Unicode-строку. Тогда, размещая строку в области данных программы, компилятор вставит между всеми символами нулевые байты. Но возникает другая проблема — программа компилируется, только если `_UNICODE` определен. Следовательно, нужен макрос, способный избирательно ставить `L` перед строковым литералом. Эту работу выполняет макрос `_TEXT`, также содержащийся в `Tchar.h`. Если `_UNICODE` определен, `_TEXT` определяется как:

```
#define _TEXT(x) L ## x
```

В ином случае `_TEXT` определяется следующим образом:

```
#define _TEXT(x) x
```

Используя этот макрос, перепишем злополучный оператор так, чтобы его можно было корректно компилировать независимо от того, определен `_UNICODE` или нет:

```
TCHAR *szError = _TEXT("Error");
```

Макрос `_TEXT` применяется и для символьных литералов. Например, чтобы проверить, является ли первый символ строки заглавной буквой `J`:

```
if (szError[0] == _TEXT('J')) {
    // первый символ - J
    :
} else {
    // первый символ - не J
    :
}
```

Типы данных, определенные в Windows для Unicode

В заголовочных файлах Windows определены следующие типы данных.

Тип данных	Описание
WCHAR	Unicode-символ
PWSTR	Указатель на Unicode-строку
PCWSTR	Указатель на строковую константу в Unicode

Эти типы данных относятся исключительно к символам и строкам в кодировке Unicode. В заголовочных файлах Windows определены также универсальные (ANSI/

Unicode) типы данных PTSTR и PCTSTR, указывающие — в зависимости от того, определен ли при компиляции макрос UNICODE, — на ANSI- или на Unicode-строку.

Кстати, на этот раз имя макроя UNICODE не предваряется знаком подчеркивания. Дело в том, что макроя _UNICODE используется в заголовочных файлах библиотеки C, а макроя UNICODE — в заголовочных файлах Windows. Для компиляции модулей исходного кода обычно приходится определять оба макроя.

Unicode- и ANSI-функции в Windows

Я уже говорил, что существует две функции *CreateWindowEx*: одна принимает строки в Unicode, другая — в ANSI. Все так, но в действительности прототипы этих функций чуть-чуть отличаются:

```
HWND WINAPI CreateWindowExW(
    DWORD dwExStyle,
    PCWSTR pClassName,
    PCWSTR pWindowName,
    DWORD dwStyle,
    int X,
    int Y,
    int nWidth,
    int nHeight,
    HWND hWndParent,
    HMENU hMenu,
    HINSTANCE hInstance,
    PVOID pParam);
```

```
HWND WINAPI CreateWindowExA(
    DWORD dwExStyle,
    PCSTR pClassName,
    PCSTR pWindowName,
    DWORD dwStyle,
    int X,
    int Y,
    int nWidth,
    int nHeight,
    HWND hWndParent,
    HMENU hMenu,
    HINSTANCE hInstance,
    PVOID pParam);
```

CreateWindowExW — это Unicode-версия. Буква *W* в конце имени функции — аббревиатура слова *wide* (широкий). Символы Unicode занимают по 16 битов каждый, поэтому их иногда называют широкими символами (wide characters). Буква *A* в конце имени *CreateWindowExA* указывает, что данная версия функции принимает ANSI-строки.

Но обычно *CreateWindowExW* или *CreateWindowExA* напрямую не вызывают, а обращаются к *CreateWindowEx* — макроя, определенному в файле WinUser.h:

```
#ifdef UNICODE
#define CreateWindowEx CreateWindowExW
#else
#define CreateWindowEx CreateWindowExA
#endif // !UNICODE
```

Какая именно версия *CreateWindowEx* будет вызвана, зависит от того, определен ли UNICODE в период компиляции. Перенося 16-разрядное Windows-приложение на платформу Win32, Вы, вероятно, не станете определять UNICODE. Тогда все вызовы *CreateWindowEx* будут преобразованы в вызовы *CreateWindowExA* — ANSI-версии функций. И перенос приложения упростится, ведь 16-разрядная Windows работает только с ANSI-версией *CreateWindowEx*.

В Windows 2000 функция *CreateWindowExA* — просто шлюз (транслятор), который выделяет память для преобразования строк из ANSI в Unicode и вызывает *CreateWindowExW*, передавая ей преобразованные строки. Когда *CreateWindowExW* вернет управление, *CreateWindowExA* освободит буферы и передаст Вам описатель окна.

Разрабатывая DLL, которую будут использовать и другие программисты, предусматривайте в ней по две версии каждой функции — для ANSI и для Unicode. В ANSI-версии просто выделяйте память, преобразуйте строки и вызывайте Unicode-версию той же функции. (Этот процесс я продемонстрирую позже.)

В Windows 98 основную работу выполняет *CreateWindowExA*. В этой операционной системе предусмотрены точки входа для всех функций Windows, принимающих Unicode-строки, но функции не транслируют их в ANSI, а просто сообщают об ошибке. Последующий вызов *GetLastError* дает ERROR_CALL_NOT_IMPLEMENTED. Должным образом действуют только ANSI-версии функций. Ваше приложение не будет работать в Windows 98, если в скомпилированном коде присутствуют вызовы «широкосимвольных» функций.

Некоторые функции Windows API (например, *WinExec* или *OpenFile*) существуют только для совместимости с 16-разрядными программами, и их надо избегать. Лучше заменить все вызовы *WinExec* и *OpenFile* вызовами *CreateProcess* и *CreateFile* соответственно. Тем более, что старые функции просто обращаются к новым. Самая серьезная проблема с ними в том, что они не принимают строки в Unicode, — при их вызове Вы должны передавать строки в ANSI. С другой стороны, в Windows 2000 у всех новых или пока не устаревших функций обязательно есть как ANSI-, так и Unicode-версия.

Строковые функции Windows

Windows предлагает внушительный набор функций, работающих со строками. Они похожи на строковые функции из библиотеки C, например на *strcpy* и *wcscpy*. Однако функции Windows являются частью операционной системы, и многие ее компоненты используют именно их, а не аналоги из библиотеки C. Я советую отдать предпочтение функциям операционной системы. Это немножко повысит быстродействие Вашего приложения. Дело в том, что к ним часто обращаются такие тяжеловесные процессы, как оболочка операционной системы (Explorer.exe), и скорее всего эти функции будут загружены в память еще до запуска Вашего приложения.

Данные функции доступны в Windows 2000 и Windows 98. Но Вы сможете вызывать их и в более ранних версиях Windows, если установите Internet Explorer версии 4.0 или выше.

По классической схеме именования функций в операционных системах их имена состоят из символов нижнего и верхнего регистра и выглядят так: *StrCat*, *StrChr*, *StrCmp*, *StrCpy* и т. д. Для использования этих функций включите в программу заголовочный файл ShlwApi.h. Кроме того, как я уже говорил, каждая строковая функция существует в двух версиях — для ANSI и для Unicode (например, *StrCatA* и *StrCatW*). Поскольку это функции операционной системы, их имена автоматически преобразуются в нужную форму, если в исходном тексте Вашей программы перед ее сборкой определен UNICODE.

Создание программ, способных использовать и ANSI, и Unicode

Неплохая мысль — заранее подготовить свое приложение к Unicode, даже если Вы пока не планируете работать с этой кодировкой. Вот главное, что для этого нужно:

- привыкайте к тому, что текстовые строки — это массивы символов, а не массивы байтов или значений типа *char*;
- используйте универсальные типы данных (вроде TCHAR или PTSTR) для текстовых символов и строк;
- используйте явные типы данных (вроде BYTE или PBYTE) для байтов, указателей на байты и буферов данных;
- применяйте макрос _TEXT для определения символьных и строковых литералов;
- предусмотрите возможность глобальных замен (например, PSTR на PTSTR);
- модифицируйте логику строковой арифметики. Например, функции обычно принимают размер буфера в символах, а не в байтах. Это значит, что вместо *sizeof(szBuffer)* Вы должны передавать (*sizeof(szBuffer)* / *sizeof(TCHAR)*). Но блок памяти для строки известной длины выделяется в байтах, а не символах, т. е. вместо *malloc(nCharacters)* нужно использовать *malloc(nCharacters * sizeof(TCHAR))*. Из всего, что я перечислил, это запомнить труднее всего — если Вы ошибетесь, компилятор не выдаст никаких предупреждений.

Разрабатывая программы-примеры для первого издания книги, я сначала написал их так, что они компилировались только с использованием ANSI. Но, дойдя до этой главы (она была тогда в конце), понял, что Unicode лучше, и решил написать примеры, которые показывали бы, как легко создавать программы, компилируемые с применением и Unicode, и ANSI. В конце концов я преобразовал все программы-примеры так, чтобы их можно было компилировать в расчете на любой из этих стандартов.

Конверсия всех программ заняла примерно 4 часа — неплохо, особенно если учсть, что у меня совсем не было опыта в этом деле.

В Windows есть набор функций для работы с Unicode-строками. Эти функции перечислены ниже.

Функция	Описание
<i>lstrcmp</i>	Выполняет конкатенацию строк
<i>lstrcmpi</i>	Сравнивает две строки с учетом регистра букв
<i>lstrcmpi</i>	Сравнивает две строки без учета регистра букв
<i>lstrcpy</i>	Копирует строку в другой участок памяти
<i>lstrlen</i>	Возвращает длину строки в символах

Они реализованы как макросы, вызывающие либо Unicode-, либо ANSI-версию функции в зависимости от того, определен ли UNICODE при компиляции исходного модуля. Например, если UNICODE не определен, *lstrcmp* раскрывается в *lstrcmpA*, определен — в *lstrcmpW*.

Строковые функции *lstrcmp* и *lstrcmpi* ведут себя не так, как их аналоги из библиотеки C (*strcmp*, *strncpy*, *wcsstrcmp* и *wcsncpy*), которые просто сравнивают кодовые позиции в символах строк. Игнорируя фактические символы, они сравнивают числовое значение каждого символа первой строки с числовым значением символа второй

строки. Но *lstrcmp* и *lstrcmpi* реализованы через вызовы Windows-функции *CompareString*:

```
int CompareString(
    LCID lcid,
    DWORD fdwStyle,
    PCWSTR pString1,
    int cch1,
    PCWSTR pString2,
    int cch2);
```

Она сравнивает две Unicode-строки. Первый параметр задает так называемый идентификатор локализации (locale ID, LCID) — 32-битное значение, определяющее конкретный язык. С помощью этого идентификатора *CompareString* сравнивает строки с учетом значения конкретных символов в данном языке. Так что она действует куда осмысленнее, чем функции библиотеки С.

Когда любая из функций семейства *lstrcmp* вызывает *CompareString*, в первом параметре передается результат вызова Windows-функции *GetThreadLocale*:

```
LCID GetThreadLocale();
```

Она возвращает уже упомянутый идентификатор, который назначается потоку в момент его создания.

Второй параметр функции *CompareString* указывает флаги, модифицирующие метод сравнения строк. Допустимые флаги перечислены в следующей таблице.

Флаг	Действие
NORM_IGNORECASE	Различия в регистре букв игнорируются
NORM_IGNOREKANATYPE	Различия между знаками хираганы и катаканы игнорируются
NORM_IGNORENONSPACE	Знаки, отличные от пробелов, игнорируются
NORM_IGNORESYMBOLS	Символы, отличные от алфавитно-цифровых, игнорируются
NORM_IGNOREWIDTH	Разница между одно- и двухбайтовым представлением одного и того же символа игнорируется
SORT_STRINGSORT	Знаки препинания обрабатываются так же, как и символы, отличные от алфавитно-цифровых

Вызывая *CompareString*, функция *lstrcmp* передает в параметре *fdwStyle* нуль, а *lstrcmpi* — флаг NORM_IGNORECASE. Остальные четыре параметра определяют две строки и их длину. Если *cch1* равен –1, функция считает, что строка *pString1* завершается нулевым символом, и автоматически вычисляет ее длину. То же относится и к параметрам *cch2* и *pString2*.

Многие функции С-библиотеки с Unicode-строками толком не работают. Так, *tolower* и *toupper* неправильно преобразуют регистр букв со знаками ударения. Поэтому для Unicode-строк лучше использовать соответствующие Windows-функции. К тому же они корректно работают и с ANSI-строками.

Первые две функции:

```
PTSTR CharLower(PTSTR pszString);
```

```
PTSTR CharUpper(PTSTR pszString);
```

преобразуют либо отдельный символ, либо целую строку с нулевым символом в конце. Чтобы преобразовать всю строку, просто передайте ее адрес. Но, преобразуя отдельный символ, Вы должны передать его так:

```
TCHAR cLowerCaseChr = CharLower((PTSTR) szString[0]);
```

Приведение типа отдельного символа к PTSTR вызывает обнуление старших 16 битов передаваемого значения, а в его младшие 16 битов помещается сам символ. Обнаружив, что старшие 16 битов этого значения равны 0, функция «поймет», что Вы хотите преобразовать не строку, а отдельный символ. Возвращаемое 32-битное значение содержит результат преобразования в младших 16 битах.

Следующие две функции аналогичны двум предыдущим за исключением того, что они преобразуют символы, содержащиеся в буфере (который не требуется завершать нулевым символом):

```
DWORD CharLowerBuff(  
    PTSTR pszString,  
    DWORD cchString);
```

```
DWORD CharUpperBuff(  
    PTSTR pszString,  
    DWORD cchString);
```

Прочие функции библиотеки С (например, *isalpha*, *islower* и *isupper*) возвращают значение, которое сообщает, является ли данный символ буквой, а также строчная она или прописная. В Windows API тоже есть подобные функции, но они учитывают и язык, выбранный пользователем в Control Panel:

```
BOOL IsCharAlpha(TCHAR ch);  
BOOL IsCharAlphaNumeric(TCHAR ch);  
BOOL IsCharLower(TCHAR ch);  
BOOL IsCharUpper(TCHAR ch);
```

И последняя группа функций из библиотеки С, о которых я хотел рассказать, — *printf*. Если при компиляции _UNICODE определен, они ожидают передачи всех символьных и строковых параметров в Unicode; в ином случае — в ANSI.

Microsoft ввела в семейство функций *printf* своей С-библиотеки дополнительные типы полей, часть из которых не поддерживается в ANSI C. Они позволяют легко сравнивать и смещивать символы и строки с разной кодировкой. Также расширена функция *wsprintf* операционной системы. Вот несколько примеров (обратите внимание на использование буквы *s* в верхнем и нижнем регистре):

```
char szA[100]; // строковый буфер в ANSI  
WCHAR szW[100]; // строковый буфер в Unicode  
  
// обычный вызов sprintf: все строки в ANSI  
sprintf(szA, "%s", "ANSI Str");  
  
// преобразуем строку из Unicode в ANSI  
sprintf(szA, "%S", L"Unicode Str");  
  
// обычный вызов swprintf: все строки в Unicode  
swprintf(szW, L"%s", L"Unicode Str");  
  
// преобразуем строку из ANSI в Unicode  
swprintf(szW, L"%S", "ANSI Str");
```

Ресурсы

Компилятор ресурсов генерирует двоичное представление всех ресурсов, используемых Вашей программой. Строки в ресурсах (таблицы строк, шаблоны диалоговых окон, меню и др.) всегда записываются в Unicode. Если в программе не определяется макрос UNICODE, Windows 98 и Windows 2000 сами проводят нужные преобразования. Например, если при компиляции исходного модуля UNICODE не определен, вызов *LoadString* на самом деле приводит к вызову *LoadStringA*, которая читает строку из ресурсов и преобразует ее в ANSI. Затем Вашей программе возвращается ANSI-представление строки.

Текстовые файлы

Текстовых файлов в кодировке Unicode пока очень мало. Ни в одном текстовом файле, поставляемом с операционными системами или другими программными продуктами Microsoft, не используется Unicode. Думаю, однако, что эта тенденция изменится в будущем — пусть даже в отдаленном. Например, программа Notepad в Windows 2000 позволяет создавать или открывать как Unicode-, так и ANSI-файлы. Посмотрите на ее диалоговое окно Save As (рис. 2-2) и обратите внимание на предлагаемые форматы текстовых файлов.

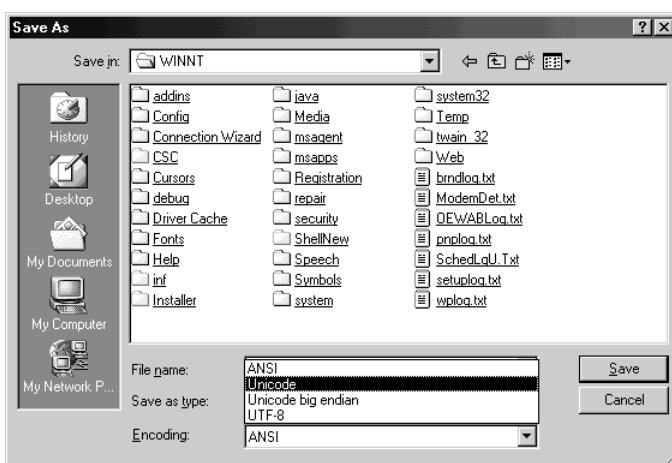


Рис. 2-2. Диалоговое окно Save As программы Notepad в Windows 2000

Многим приложениям, которые открывают и обрабатывают текстовые файлы (например, компиляторам), было бы удобнее, если после открытия файла можно было бы определить, содержит он символы в ANSI или в Unicode. В этом может помочь функция *IsTextUnicode*:

```
DWORD IsTextUnicode(CONST PVOID pvBuffer, int cb, PINT pResult);
```

Проблема с текстовыми файлами в том, что не существует четких и строгих правил относительно их содержимого. Это крайне затрудняет определение того, содержит файл символы в ANSI или в Unicode. Поэтому *IsTextUnicode* применяет набор статистических и детерминистских методов для того, чтобы сделать взвешенное предположение о содержимом буфера. Поскольку тут больше алхимии, чем точной науки, нет гарантий, что Вы не получите неверные результаты от *IsTextUnicode*.

Первый ее параметр, *pvBuffer*, указывает на буфер, подлежащий проверке. При этом используется указатель типа void, поскольку неизвестно, в какой кодировке данный массив символов.

Параметр *cb* определяет число байтов в буфере, на который указывает *pvBuffer*. Так как содержимое буфера не известно, *cb* — счетчик именно байтов, а не символов. Заметьте: вовсе не обязательно задавать всю длину буфера. Но чем больше байтов проанализирует функция, тем больше шансов получить правильный результат.

Параметр *pResult* — это адрес целочисленной переменной, которую надо инициализировать перед вызовом функции. Ее значение сообщает, какие тесты должна пройти *IsUnicode*. Если *pResult* равен NULL, функция *IsUnicode* делает все проверки. (Подробнее об этом см. документацию Platform SDK.)

Функция возвращает TRUE, если считает, что буфер содержит текст в Unicode, и FALSE — в ином случае. Да-да, она возвращает именно булево значение, хотя в прототипе указано DWORD. Если через целочисленную переменную, на которую указывает *pResult*, были запрошены лишь определенные тесты, функция (перед возвратом управления) устанавливает ее биты в соответствии с результатами этих тестов.

WINDOWS 98 В Windows 98 функция *IsUnicode* по сути не реализована и просто возвращает FALSE; последующий вызов *GetLastError* дает код ошибки ERROR_CALL_NOT_IMPLEMENTED.

Применение функции *IsUnicode* иллюстрирует программа-пример FileRev (см. главу 17).

Перекодировка строк из Unicode в ANSI и обратно

Windows-функция *MultiByteToWideChar* преобразует мультибайтовые символы строки в «широкобайтовые»:

```
int MultiByteToWideChar(
    UINT uCodePage,
    DWORD dwFlags,
    PCSTR pMultiByteStr,
    int cchMultiByte,
    PWSTR pWideCharStr,
    int cchWideChar);
```

Параметр *uCodePage* задает номер кодовой страницы, связанной с мультибайтовой строкой. Параметр *dwFlags* влияет на преобразование букв с диакритическими знаками. Обычно эти флаги не используются, и *dwFlags* равен 0. Параметр *pMultiByteStr* указывает на преобразуемую строку, а *cchMultiByte* определяет ее длину в байтах. Функция самостоятельно определяет длину строки, если *cchMultiByte* равен –1.

Unicode-версия строки, полученная в результате преобразования, записывается в буфер по адресу, указанному в *pWideCharStr*. Максимальный размер этого буфера (в символах) задается в параметре *cchWideChar*. Если он равен 0, функция ничего не преобразует, а просто возвращает размер буфера, необходимого для сохранения результата преобразования. Обычно конверсия мультибайтовой строки в ее Unicode-эквивалент проходит так:

1. Вызывают *MultiByteToWideChar*, передавая NULL в параметре *pWideCharStr* и 0 в параметре *cchWideChar*.
2. Выделяют блок памяти, достаточный для сохранения преобразованной строки. Его размер получают из предыдущего вызова *MultiByteToWideChar*.

3. Снова вызывают *MultiByteToWideChar*, на этот раз передавая адрес выделенного буфера в параметре *pWideCharStr*, а размер буфера, полученный при первом обращении к этой функции, — в параметре *cchWideChar*.
4. Работают с полученной строкой.
5. Освобождают блок памяти, занятый Unicode-строкой.

Обратное преобразование выполняет функция *WideCharToMultiByte*:

```
int WideCharToMultiByte(
    UINT uCodePage,
    DWORD dwFlags,
    PCWSTR pWideCharStr,
    int cchWideChar,
    PSTR pMultiByteStr,
    int cchMultiByte,
    PCSTR pDefaultChar,
    PBOOL pfUsedDefaultChar);
```

Она очень похожа на *MultiByteToWideChar*. И опять *uCodePage* определяет кодовую страницу для строки — результата преобразования. Дополнительный контроль над процессом преобразования дает параметр *dwFlags*. Его флаги влияют на символы с диакритическими знаками и на символы, которые система не может преобразовать. Такой уровень контроля обычно не нужен, и *dwFlags* приравнивается 0.

Параметр *pWideCharStr* указывает адрес преобразуемой строки, а *cchWideChar* задает ее длину в символах. Функция сама определяет длину исходной строки, если *cchWideChar* равен –1.

Мультибайтовый вариант строки, полученный в результате преобразования, записывается в буфер, на который указывает *pMultiByteStr*. Параметр *cchMultiByte* определяет максимальный размер этого буфера в байтах. Передав нулевое значение в *cchMultiByte*, Вы заставите функцию сообщить размер буфера, требуемого для записи результата. Обычно конверсия широкобайтовой строки в мультибайтовую проходит в той же последовательности, что и при обратном преобразовании.

Очевидно, Вы заметили, что *WideCharToMultiByte* принимает на два параметра больше, чем *MultiByteToWideChar*; это *pDefaultChar* и *pfUsedDefaultChar*. Функция *WideCharToMultiByte* использует их, только если встречает широкий символ, не представленный в кодовой странице, на которую ссылается *uCodePage*. Если его преобразование невозможно, функция берет символ, на который указывает *pDefaultChar*. Если этот параметр равен NULL (как обычно и бывает), функция использует системный символ по умолчанию. Таким символом обычно служит знак вопроса, что при операциях с именами файлов очень опасно, поскольку он является и символом подстановки.

Параметр *pfUsedDefaultChar* указывает на переменную типа BOOL, которую функция устанавливает как TRUE, если хоть один символ из широкосимвольной строки не преобразован в свой мультибайтовый эквивалент. Если же все символы преобразованы успешно, функция устанавливает переменную как FALSE. Обычно Вы передаете NULL в этом параметре.

Подробнее эти функции и их применение описаны в документации Platform SDK.

Эти две функции позволяют легко создавать ANSI- и Unicode-версии других функций, работающих со строками. Например, у Вас есть DLL, содержащая функцию, которая переставляет все символы строки в обратном порядке. Unicode-версию этой функции можно было бы написать следующим образом.

```
BOOL StringReverseW(PWSTR pWideCharStr) {  
  
    // получаем указатель на последний символ в строке  
    PWSTR pEndOfStr = pWideCharStr + wcslen(pWideCharStr) - 1;  
    wchar_t cCharT;  
    // повторяем, пока не дойдем до середины строки  
    while (pWideCharStr < pEndOfStr) {  
        // записываем символ во временную переменную  
        cCharT = *pWideCharStr;  
  
        // помещаем последний символ на место первого  
        *pWideCharStr = *pEndOfStr;  
  
        // копируем символ из временной переменной на место  
        // последнего символа  
        *pEndOfStr = cCharT;  
  
        // продвигаемся на 1 символ вправо  
        pWideCharStr++;  
  
        // продвигаемся на 1 символ влево  
        pEndOfStr--;  
    }  
  
    // строка обращена; сообщаем об успешном завершении  
    return(TRUE);  
}
```

ANSI-версию этой функции можно написать так, чтобы она вообще ничем не занималась, а просто преобразовывала ANSI-строку в Unicode, передавала ее в функцию *StringReverseW* и конвертировала обращенную строку снова в ANSI. Тогда функция должна выглядеть примерно так:

```
BOOL StringReverseA(PSTR pMultiByteStr) {  
    PWSTR pWideCharStr;  
    int nLenOfWideCharStr;  
    BOOL fOk = FALSE;  
  
    // вычисляем количество символов, необходимых  
    // для хранения широкосимвольной версии строки  
    nLenOfWideCharStr = MultiByteToWideChar(CP_ACP, 0,  
        pMultiByteStr, -1, NULL, 0);  
  
    // Выделяем память из стандартной кучи процесса,  
    // достаточную для хранения широкосимвольной строки.  
    // Не забудьте, что MultiByteToWideChar возвращает  
    // количество символов, а не байтов, поэтому мы должны  
    // умножить это число на размер широкого символа.  
    pWideCharStr = HeapAlloc(GetProcessHeap(), 0,  
        nLenOfWideCharStr * sizeof(WCHAR));  
  
    if (pWideCharStr == NULL)  
        return(fOk);
```

```
// преобразуем мультибайтовую строку в широкосимвольную
MultiByteToWideChar(CP_ACP, 0, pMultiByteStr, -1,
    pWideCharStr, nLenOfWideCharStr);

// вызываем широкосимвольную версию этой функции
// для выполнения настоящей работы
fOk = StringReverseW(pWideCharStr);

if (fOk) {
    // преобразуем широкосимвольную строку обратно в мультибайтовую
    WideCharToMultiByte(CP_ACP, 0, pWideCharStr, -1,
        pMultiByteStr, strlen(pMultiByteStr), NULL, NULL);
}

// освобождаем память, выделенную под широкобайтовую строку
HeapFree(GetProcessHeap(), 0, pWideCharStr);

return(fOk);
}
```

И, наконец, в заголовочном файле, поставляемом вместе с DLL, прототипы этих функций были бы такими:

```
BOOL StringReverseW (PWSTR pWideCharStr);
BOOL StringReverseA (PSTR pMultiByteStr);

#ifndef UNICODE
#define StringReverse StringReverseW
#else
#define StringReverse StringReverseA
#endif // !UNICODE
```

Объекты ядра

Изучение Windows API мы начнем с объектов ядра и их описателей (handles). Эта глава посвящена сравнительно абстрактным концепциям, т. е. мы, не углубляясь в специфику тех или иных объектов ядра, рассмотрим их общие свойства.

Я бы предпочел начать с чего-то более конкретного, но без четкого понимания объектов ядра Вам не стать настоящим профессионалом в области разработки Windows-программ. Эти объекты используются системой и нашими приложениями для управления множеством самых разных ресурсов: процессами, потоками, файлами и т. д. Концепции, представленные здесь, будут встречаться на протяжении всей книги. Однако я прекрасно понимаю, что часть материалов не уляжется у Вас в голове до тех пор, пока Вы не приступите к работе с объектами ядра, используя реальные функции. И при чтении последующих глав книги Вы, наверное, будете время от времени возвращаться к этой главе.

Что такое объект ядра

Создание, открытие и прочие операции с объектами ядра станут для Вас, как разработчика Windows-приложений, повседневной рутиной. Система позволяет создавать и оперировать с несколькими типами таких объектов, в том числе: маркерами доступа (access token objects), файлами (file objects), проекциями файлов (file-mapping objects), портами завершения ввода-вывода (I/O completion port objects), заданиями (job objects), почтовыми ящиками (mailslot objects), мьютексами (mutex objects), каналами (pipe objects), процессами (process objects), семафорами (semaphore objects), потоками (thread objects) и ожидающими таймерами (waitable timer objects). Эти объекты создаются Windows-функциями. Например, *CreateFileMapping* заставляет систему сформировать объект «проекция файла». Каждый объект ядра — на самом деле просто блок памяти, выделенный ядром и доступный только ему. Этот блок представляет собой структуру данных, в элементах которой содержится информация об объекте. Некоторые элементы (дескриптор защиты, счетчик числа пользователей и др.) присутствуют во всех объектах, но большая их часть специфична для объектов конкретного типа. Например, у объекта «процесс» есть идентификатор, базовый приоритет и код завершения, а у объекта «файл» — смещение в байтах, режим разделения и режим открытия.

Поскольку структуры объектов ядра доступны только ядру, приложение не может самостоятельно найти эти структуры в памяти и напрямую модифицировать их содержимое. Такое ограничение Microsoft ввела намеренно, чтобы ни одна программа не нарушила целостность структур объектов ядра. Это же ограничение позволяет Microsoft вводить, убирать или изменять элементы структур, не нарушая работы каких-либо приложений.

Но вот вопрос: если мы не можем напрямую модифицировать эти структуры, то как же наши приложения оперируют с объектами ядра? Ответ в том, что в Windows

предусмотрен набор функций, обрабатывающих структуры объектов ядра по строго определенным правилам. Мы получаем доступ к объектам ядра только через эти функции. Когда Вы вызываете функцию, создающую объект ядра, она возвращает описатель, идентифицирующий созданный объект. Описатель следует рассматривать как «непрозрачное» значение, которое может быть использовано любым потоком Вашего процесса. Этот описатель Вы передаете Windows-функциям, сообщая системе, какой объект ядра Вас интересует. Но об описателях мы поговорим позже (в этой главе).

Для большей надежности операционной системы Microsoft сделала так, чтобы значения описателей зависели от конкретного процесса. Поэтому, если Вы передадите такое значение (с помощью какого-либо механизма межпроцессной связи) потоку другого процесса, любой вызов из того процесса со значением описателя, полученного в Вашем процессе, даст ошибку. Но не волнуйтесь, в конце главы мы рассмотрим три механизма корректного использования несколькими процессами одного объекта ядра.

Учет пользователей объектов ядра

Объекты ядра принадлежат ядру, а не процессу. Иначе говоря, если Ваш процесс вызывает функцию, создающую объект ядра, а затем завершается, объект ядра может быть не разрушен. В большинстве случаев такой объект все же разрушается; но если созданный Вами объект ядра используется другим процессом, ядро запретит разрушение объекта до тех пор, пока от него не откажется и тот процесс.

Ядру известно, сколько процессов использует конкретный объект ядра, поскольку в каждом объекте есть счетчик числа его пользователей. Этот счетчик — один из элементов данных, общих для всех типов объектов ядра. В момент создания объекта счетчику присваивается 1. Когда к существующему объекту ядра обращается другой процесс, счетчик увеличивается на 1. А когда какой-то процесс завершается, счетчики всех используемых им объектов ядра автоматически уменьшаются на 1. Как только счетчик какого-либо объекта обнуляется, ядро уничтожает этот объект.

Защита

Объекты ядра можно защитить дескриптором защиты (security descriptor), который описывает, кто создал объект и кто имеет права на доступ к нему. Дескрипторы защиты обычно используют при написании серверных приложений; создавая клиентское приложение, Вы можете игнорировать это свойство объектов ядра.

WINDOWS 98 В Windows 98 дескрипторы защиты отсутствуют, так как она не предназначена для выполнения серверных приложений. Тем не менее Вы должны знать о тонкостях, связанных с защитой, и реализовать соответствующие механизмы, чтобы Ваше приложение корректно работало и в Windows 2000.

Почти все функции, создающие объекты ядра, принимают указатель на структуру **SECURITY_ATTRIBUTES** как аргумент, например:

```
HANDLE CreateFileMapping(
    HANDLE hFile,
    PSECURITY_ATTRIBUTES psa,
    DWORD f1Protect,
    DWORD dwMaximumSizeHigh,
    DWORD dwMaximumSizeLow,
    PCTSTR pszName);
```

Большинство приложений вместо этого аргумента передает NULL и создает объект с защитой по умолчанию. Такая защита подразумевает, что создатель объекта и любой член группы администраторов получают к нему полный доступ, а все прочие к объекту не допускаются. Однако Вы можете создать и инициализировать структуру SECURITY_ATTRIBUTES, а затем передать ее адрес. Она выглядит так:

```
typedef struct _SECURITY_ATTRIBUTES {  
    DWORD nLength;  
    LPVOID lpSecurityDescriptor;  
    BOOL bInheritHandle;  
} SECURITY_ATTRIBUTES;
```

Хотя структура называется SECURITY_ATTRIBUTES, лишь один ее элемент имеет отношение к защите — *lpSecurityDescriptor*. Если надо ограничить доступ к созданному Вами объекту ядра, создайте дескриптор защиты и инициализируйте структуру SECURITY_ATTRIBUTES следующим образом:

```
SECURITY_ATTRIBUTES sa;  
sa.nLength = sizeof(sa);           // используется для выяснения версий  
sa.lpSecurityDescriptor = pSD;     // адрес инициализированной SD  
sa.bInheritHandle = FALSE;         // об этом позже  
HANDLE hFileMapping = CreateFileMapping(INVALID_HANDLE_VALUE, &sa,  
    PAGE_READWRITE, 0, 1024, "MyFileMapping");  
:
```

Рассмотрение элемента *bInheritHandle* я отложу до раздела о наследовании, так как этот элемент не имеет ничего общего с защитой.

Желая получить доступ к существующему объекту ядра (вместо того чтобы создавать новый), укажите, какие операции Вы намерены проводить над объектом. Например, если бы я захотел считывать данные из существующей проекции файла, то вызвал бы функцию *OpenFileMapping* таким образом:

```
HANDLE hFileMapping = OpenFileMapping(FILE_MAP_READ, FALSE, "MyFileMapping");
```

Передавая FILE_MAP_READ первым параметром в функцию *OpenFileMapping*, я сообщаю, что, как только мне предоставят доступ к проекции файла, я буду считывать из нее данные. Функция *OpenFileMapping*, прежде чем вернуть действительный описатель, проверяет тип защиты объекта. Если меня, как зарегистрировавшегося пользователя, допускают к существующему объекту ядра «проекция файла», *OpenFileMapping* возвращает действительный описатель. Но если мне отказывают в доступе, *OpenFileMapping* возвращает NULL, а вызов *GetLastError* дает код ошибки 5 (или ERROR_ACCESS_DENIED). Но опять же, в основной массе приложений защиту не используют, и поэтому я больше не буду задерживаться на этой теме.

WINDOWS 98 Хотя в большинстве приложений нет нужды беспокоиться о защите, многие функции Windows требуют, чтобы Вы передавали им информацию о нужном уровне защиты. Некоторые приложения, написанные для Windows 98, в Windows 2000 толком не работают из-за того, что при их реализации не было уделено должного внимания защите.

Представьте, что при запуске приложение считывает данные из какого-то раздела реестра. Чтобы делать это корректно, оно должно вызывать функцию *RegOpenKeyEx*, передавая значение KEY_QUERY_VALUE, которое разрешает операцию чтения в указанном разделе.

Однако многие приложения для Windows 98 создавались без учета специфики Windows 2000. Поскольку Windows 98 не защищает свой реестр, разработчики часто вызывали *RegOpenKeyEx* со значением KEY_ALL_ACCESS. Так проще и не надо ломать голову над тем, какой уровень доступа требуется на самом деле. Но проблема в том, что раздел реестра может быть доступен для чтения и блокирован для записи. В Windows 2000 вызов *RegOpenKeyEx* со значением KEY_ALL_ACCESS заканчивается неудачно, и без соответствующего контроля ошибок приложение может повести себя совершенно непредсказуемо.

Если бы разработчик хоть немного подумал о защите и поменял значение KEY_ALL_ACCESS на KEY_QUERY_VALUE (только-то и всего!), его продукт мог бы работать в обеих операционных системах.

Пренебрежение флагами, определяющими уровень доступа, — одна из самых крупных ошибок, совершаемых разработчиками. Правильное их использование позволило бы легко перенести многие приложения Windows 98 в Windows 2000.

Кроме объектов ядра Ваша программа может использовать объекты других типов — меню, окна, курсоры мыши, кисти и шрифты. Они относятся к объектам User или GDI. Новичок в программировании для Windows может запутаться, пытаясь отличить объекты User или GDI от объектов ядра. Как узнать, например, чьим объектом — User или ядра — является данный значок? Выяснить, не принадлежит ли объект ядру, проще всего так: проанализировать функцию, создающую объект. Практически у всех функций, создающих объекты ядра, есть параметр, позволяющий указать атрибуты защиты, — как у *CreateFileMapping*.

В то же время у функций, создающих объекты User или GDI, нет параметра типа PSECURITY_ATTRIBUTES, и пример тому — функция *CreateIcon*:

```
HICON CreateIcon(
    HINSTANCE hinst,
    int nWidth,
    int nHeight,
    BYTE cPlanes,
    BYTE cBitsPixel,
    CONST BYTE *pbANDbits,
    CONST BYTE *pbXORbits);
```

Таблица описателей объектов ядра

При инициализации процесса система создает в нем таблицу описателей, используемую только для объектов ядра. Сведения о структуре этой таблицы и управлении ею незадокументированы. Вообще-то я воздерживаюсь от рассмотрения недокументированных частей операционных систем. Но в данном случае стоит сделать исключение — квалифицированный Windows-программист, на мой взгляд, должен понимать, как устроена таблица описателей в процессе. Поскольку информация о таблице описателей незадокументирована, я не ручаюсь за ее стопроцентную достоверность и к тому же эта таблица по-разному реализуется в Windows 2000, Windows 98 и Windows CE. Таким образом, следующие разделы помогут понять, что представляет собой таблица описателей, но вот что система действительно делает с ней — этот вопрос я оставляю открытым.

В таблице 3-1 показано, как выглядит таблица описателей, принадлежащая процессу. Как видите, это просто массив структур данных. Каждая структура содержит указатель на какой-нибудь объект ядра, маску доступа и некоторые флаги.

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0x????????	0x????????	0x????????
2	0x????????	0x????????	0x????????
...

Таблица 3-1. Структура таблицы описателей, принадлежащей процессу

Создание объекта ядра

Когда процесс инициализируется в первый раз, таблица описателей еще пуста. Но стоит одному из его потоков вызвать функцию, создающую объект ядра (например, *CreateFileMapping*), как ядро выделяет для этого объекта блок памяти и инициализирует его; далее ядро просматривает таблицу описателей, принадлежащую данному процессу, и отыскивает свободную запись. Поскольку таблица еще пуста, ядро обнаруживает структуру с индексом 1 и инициализирует ее. Указатель устанавливается на внутренний адрес структуры данных объекта, маска доступа — на доступ без ограничений и, наконец, определяется последний компонент — флаги. (О флагах мы поговорим позже, в разделе о наследовании.)

Вот некоторые функции, создающие объекты ядра (список ни в коей мере на полноту не претендует):

```
HANDLE CreateThread(
    PSECURITY_ATTRIBUTES psa,
    DWORD dwStackSize,
    PTHREAD_START_ROUTINE pfnStartAddr,
    PVOID pvParam,
    DWORD dwCreationFlags,
    PDWORD pdwThreadId);

HANDLE CreateFile(
    PCTSTR pszFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    PSECURITY_ATTRIBUTES psa,
    DWORD dwCreationDistribution,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile);

HANDLE CreateFileMapping(
    HANDLE hFile,
    PSECURITY_ATTRIBUTES psa,
    DWORD flProtect,
    DWORD dwMaximumSizeHigh,
    DWORD dwMaximumSizeLow,
    PCTSTR pszName);

HANDLE CreateSemaphore(
    PSECURITY_ATTRIBUTES psa,
```

```
LONG lInitialCount,
LONG lMaximumCount,
PCTSTR pszName);
```

Все функции, создающие объекты ядра, возвращают описатели, которые привязаны к конкретному процессу и могут быть использованы в любом потоке данного процесса. Значение описателя представляет собой индекс в таблице описателей, принадлежащей процессу, и таким образом идентифицирует место, где хранится информация, связанная с объектом ядра. Вот поэтому при отладке своего приложения и просмотре фактического значения описателя объекта ядра Вы и видите такие малые величины: 1, 2 и т. д. Но помните, что физическое содержимое описателей не задокументировано и может быть изменено. Кстати, в Windows 2000 это значение определяет, по сути, не индекс, а скорее байтовое смещение нужной записи от начала таблицы описателей.

Всякий раз, когда Вы вызываете функцию, принимающую описатель объекта ядра как аргумент, Вы передаете ей значение, возвращенное одной из *Create*-функций. При этом функция смотрит в таблицу описателей, принадлежащую Вашему процессу, и считывает адрес нужного объекта ядра.

Если Вы передаете неверный индекс (описатель), функция завершается с ошибкой и *GetLastError* возвращает 6 (ERROR_INVALID_HANDLE). Это связано с тем, что на самом деле описатели представляют собой индексы в таблице, их значения привязаны к конкретному процессу и недействительны в других процессах.

Если вызов функции, создающей объект ядра, оказывается неудачен, то обычно возвращается 0 (NULL). Такая ситуация возможна только при острой нехватке памяти или при наличии проблем с защитой. К сожалению, отдельные функции возвращают в таких случаях не 0, а -1 (INVALID_HANDLE_VALUE). Например, если *CreateFile* не сможет открыть указанный файл, она вернет именно INVALID_HANDLE_VALUE. Будьте очень осторожны при проверке значения, возвращаемого функцией, которая создает объект ядра. Так, для *CreateMutex* проверка на INVALID_HANDLE_VALUE бессмысленна:

```
HANDLE hMutex = CreateMutex(...);
if (hMutex == INVALID_HANDLE_VALUE) {
    // этот код никогда не будет выполнен, так как
    // при ошибке CreateMutex возвращает NULL
}
```

Точно так же бессмыслен и следующий код:

```
HANDLE hFile = CreateFile(...);
if (hFile == NULL) {
    // и этот код никогда не будет выполнен, так как
    // при ошибке CreateFile возвращает INVALID_HANDLE_VALUE (-1)
}
```

Закрытие объекта ядра

Независимо от того, как именно Вы создали объект ядра, по окончании работы с ним его нужно закрыть вызовом *CloseHandle*:

```
BOOL CloseHandle(HANDLE hobj);
```

Эта функция сначала проверяет таблицу описателей, принадлежащую вызывающему процессу, чтобы убедиться, идентифицирует ли переданный ей индекс (описа-

тель) объект, к которому этот процесс действительно имеет доступ. Если переданный индекс правилен, система получает адрес структуры данных объекта и уменьшает в этой структуре счетчик числа пользователей; как только счетчик обнулится, ядро удалит объект из памяти.

Если же описатель неверен, происходит одно из двух. В нормальном режиме выполнения процесса *CloseHandle* возвращает FALSE, а *GetLastError* — код ERROR_INVALID_HANDLE. Но при выполнении процесса в режиме отладки система просто уведомляет отладчик об ошибке.

Перед самым возвратом управления *CloseHandle* удаляет соответствующую запись из таблицы описателей: данный описатель теперь недействителен в Вашем процессе и использовать его нельзя. При этом запись удаляется независимо от того, разрушен объект ядра или нет! После вызова *CloseHandle* Вы больше не получите доступ к этому объекту ядра; но, если его счетчик не обнулен, объект остается в памяти. Тут все нормально, это означает лишь то, что объект используется другим процессом (или процессами). Когда и остальные процессы завершат свою работу с этим объектом (тоже вызвав *CloseHandle*), он будет разрушен.

А вдруг Вы забыли вызвать *CloseHandle* — будет ли утечка памяти? И да, и нет. Утечка ресурсов (тех же объектов ядра) вполне вероятна, пока процесс еще исполняется. Однако по завершении процесса операционная система гарантированно освобождает все ресурсы, принадлежавшие этому процессу, и в случае объектов ядра действует так: в момент завершения процесса просматривает его таблицу описателей и закрывает любые открытые описатели.

Совместное использование объектов ядра несколькими процессами

Время от времени возникает необходимость в разделении объектов ядра между потоками, исполняемыми в разных процессах. Причин тому может быть несколько:

- объекты «проекции файлов» позволяют двум процессам, исполняемым на одной машине, совместно использовать одни и те же блоки данных;
- почтовые ящики и именованные каналы дают возможность программам обмениваться данными с процессами, исполняемыми на других машинах в сети;
- мьютексы, семафоры и события позволяют синхронизировать потоки, исполняемые в разных процессах, чтобы одно приложение могло уведомить другое об окончании той или иной операции.

Но поскольку описатели объектов ядра имеют смысл только в конкретном процессе, разделение объектов ядра между несколькими процессами — задача весьма непростая. У Microsoft было несколько веских причин сделать описатели «процессно-независимыми», и самая главная — устойчивость операционной системы к сбоям. Если бы описатели объектов ядра были общесистемными, то один процесс мог бы запросто получить описатель объекта, используемого другим процессом, и устроить в нем (этом процессе) настоящий хаос. Другая причина — защита. Объекты ядра защищены, и процесс, прежде чем оперировать с ними, должен запрашивать разрешение на доступ к ним.

Три механизма, позволяющие процессам совместно использовать одни и те же объекты ядра, мы рассмотрим в следующем разделе.

Наследование описателя объекта

Наследование применимо, только когда процессы связаны «родственными» отношениями (родительский-дочерний). Например, родительскому процессу доступен один или несколько описателей объектов ядра, и он решает, породив дочерний процесс, передать ему по наследству доступ к своим объектам ядра. Чтобы такой сценарий наследования сработал, родительский процесс должен выполнить несколько операций.

Во-первых, еще при создании объекта ядра этот процесс должен сообщить системе, что ему нужен наследуемый описатель данного объекта. (Имейте в виду: описатели объектов ядра наследуются, но сами объекты ядра — нет.)

Чтобы создать наследуемый описатель, родительский процесс выделяет и инициализирует структуру SECURITY_ATTRIBUTES, а затем передает ее адрес требуемой *Create*-функции. Следующий код создает объект-мьютекс и возвращает его описатель:

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE; // делаем возвращаемый описатель наследуемым

HANDLE hMutex = CreateMutex(&sa, FALSE, NULL);

:
```

Этот код инициализирует структуру SECURITY_ATTRIBUTES, указывая, что объект следует создать с защитой по умолчанию (в Windows 98 это игнорируется) и что возвращаемый описатель должен быть наследуемым.

WINDOWS 98 Хотя Windows 98 не полностью поддерживает защиту, она все же поддерживает наследование и поэтому корректно обрабатывает элемент *bInheritHandle*.

А теперь перейдем к флагам, которые хранятся в таблице описателей, принадлежащей процессу. В каждой ее записи присутствует битовый флаг, сообщающий, является данный описатель наследуемым или нет. Если Вы, создавая объект ядра, передадите в параметре типа PSECURITY_ATTRIBUTES значение NULL, то получите ненаследуемый описатель, и этот флаг будет нулевым. А если элемент *bInheritHandle* равен TRUE, флагу присваивается 1.

Допустим, какому-то процессу принадлежит таблица описателей, как в таблице 3-2.

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0xF0000000	0x????????	0x00000000
2	0x00000000	(неприменим)	(неприменим)
3	0xF0000010	0x????????	0x00000001

Таблица 3-2. Таблица описателей с двумя действительными записями

Эта таблица свидетельствует, что данный процесс имеет доступ к двум объектам ядра: описатель 1 (ненаследуемый) и 3 (наследуемый).

Следующий этап — родительский процесс порождает дочерний. Это делается с помощью функции *CreateProcess*.

```
BOOL CreateProcess(
    PCTSTR pszApplicationName,
    PTSTR pszCommandLine,
    PSECURITY_ATTRIBUTES psaProcess,
    PSECURITY_ATTRIBUTES psaThread,
    BOOL bInheritHandles,
    DWORD fdwCreate,
    PVOID pvEnvironment,
    PCTSTR pszCurDir,
    PSTARTUPINFO psiStartInfo,
    PPROCESS_INFORMATION ppiProcInfo);
```

Подробно мы рассмотрим эту функцию в следующей главе, а сейчас я хочу лишь обратить Ваше внимание на параметр *bInheritHandles*. Создавая процесс, Вы обычно передаете в этом параметре FALSE, тем самым сообщая системе, что дочерний процесс не должен наследовать наследуемые описатели, зафиксированные в таблице родительского процесса. Если же Вы передаете TRUE, дочерний процесс наследует описатели родительского. Тогда операционная система создает дочерний процесс, но не дает ему немедленно начать свою работу. Сформировав в нем, как обычно, новую (пустую) таблицу описателей, она считывает таблицу родительского процесса и копирует все ее действительные записи в таблицу дочернего — причем в те же позиции. Последний факт чрезвычайно важен, так как означает, что описатели будут идентичны в обоих процессах (родительском и дочернем).

Помимо копирования записей из таблицы описателей, система увеличивает значения счетчиков соответствующих объектов ядра, поскольку эти объекты теперь используются обоими процессами. Чтобы уничтожить какой-то объект ядра, его описатель должны закрыть (вызовом *CloseHandle*) оба процесса. Кстати, сразу после возврата управления функцией *CreateProcess* родительский процесс может закрыть свой описатель объекта, и это никак не отразится на способности дочернего процесса манипулировать с этим объектом.

В таблице 3-3 показано состояние таблицы описателей в дочернем процессе — перед самым началом его исполнения. Как видите, записи 1 и 2 не инициализированы, и поэтому данные описатели неприменимы в дочернем процессе. Однако индекс 3 действительно идентифицирует объект ядра по тому же (что и в родительском) адресу 0xF0000010. При этом маска доступа и флаги в родительском и дочернем процессах тоже идентичны. Так что, если дочерний процесс в свою очередь породит новый («внука» по отношению к исходному родительскому), «внук» унаследует данный описатель объекта ядра с теми же значениями, правами доступа и флагами, а счетчик числа пользователей этого объекта ядра вновь увеличится на 1.

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0x00000000	(неприменим)	(неприменим)
2	0x00000000	(неприменим)	(неприменим)
3	0xF0000010	0x????????	0x00000001

Таблица 3-3. Таблица описателей в дочернем процессе (после того как он унаследовал от родительского один наследуемый описатель)

Наследуются только описатели объектов, существующие на момент создания дочернего процесса. Если родительский процесс создаст после этого новые объекты

ядра с наследуемыми описателями, то эти описатели будут уже недоступны дочернему процессу.

Для наследования описателей объектов характерно одно очень странное свойство: дочерний процесс не имеет ни малейшего понятия, что он унаследовал какие-то описатели. Поэтому наследование описателей объектов ядра полезно, только когда дочерний процесс сообщает, что при его создании родительским процессом он ожидает доступа к какому-нибудь объекту ядра. Тут надо заметить, что обычно родительское и дочернее приложения пишутся одной фирмой, но в принципе дочернее приложение может написать и стороннюю фирму, если в этой программе задокументировано, чего именно она ждет от родительского процесса.

Для этого в дочерний процесс обычно передают значение ожидаемого им описателя объекта ядра как аргумент в командной строке. Инициализирующий код дочернего процесса анализирует командную строку (чаще всего вызовом *sscanf*), извлекает из нее значение описателя, и дочерний процесс получает неограниченный доступ к объекту. При этом механизм наследования срабатывает только потому, что значение описателя общего объекта ядра в родительском и дочернем процессах одинаково, — и именно по этой причине родительский процесс может передать значение описателя как аргумент в командной строке.

Для наследственной передачи описателя объекта ядра от родительского процесса дочернему, конечно же, годятся и другие формы межпроцессной связи. Один из приемов заключается в том, что родительский процесс дожидается окончания инициализации дочернего (через функцию *WaitForInputIdle*, рассматриваемую в главе 9), а затем посыпает (синхронно или асинхронно) сообщение окну, созданному потоком дочернего процесса.

Еще один прием: родительский процесс добавляет в свой блок переменных окружения новую переменную. Она должна быть «узнаваема» дочерним процессом и содержать значение наследуемого описателя объекта ядра. Далее родительский процесс создает дочерний, тот наследует переменные окружения родительского процесса и, вызвав *GetEnvironmentVariable*, получает нужный описатель. Такой прием особенно хорош, когда дочерний процесс тоже порождает процессы, — ведь все переменные окружения вновь наследуются.

Изменение флагов описателя

Иногда встречаются ситуации, в которых родительский процесс создает объект ядра с наследуемым описателем, а затем порождает два дочерних процесса. Но наследуемый описатель нужен только одному из них. Иначе говоря, время от времени возникает необходимость контролировать, какой из дочерних процессов наследует описатели объектов ядра. Для этого модифицируйте флаг наследования, связанный с описателем, вызовом *SetHandleInformation*:

```
BOOL SetHandleInformation(
    HANDLE hObject,
    DWORD dwMask,
    DWORD dwFlags);
```

Как видите, эта функция принимает три параметра. Первый (*hObject*) идентифицирует допустимый описатель. Второй (*dwMask*) сообщает функции, какой флаг (или флаги) Вы хотите изменить. На сегодняшний день с каждым описателем связано два флага:

```
#define HANDLE_FLAG_INHERIT      0x00000001
#define HANDLE_FLAG_PROTECT_FROM_CLOSE 0x00000002
```

Чтобы изменить сразу все флаги объекта, нужно объединить их побитовой операцией OR.

И, наконец, третий параметр функции *SetHandleInformation* — *dwFlags* — указывает, в какое именно состояние следует перевести флаги. Например, чтобы установить флаг наследования для описателя объекта ядра:

```
SetHandleInformation(hobj, HANDLE_FLAG_INHERIT, HANDLE_FLAG_INHERIT);
```

а чтобы сбросить этот флаг:

```
SetHandleInformation(hobj, HANDLE_FLAG_INHERIT, 0);
```

Флаг HANDLE_FLAG_PROTECT_FROM_CLOSE сообщает системе, что данный описатель закрывать нельзя:

```
SetHandleInformation(hobj, HANDLE_FLAG_PROTECT_FROM_CLOSE,  
    HANDLE_FLAG_PROTECT_FROM_CLOSE);  
CloseHandle(hobj); // генерируется исключение
```

Если какой-нибудь поток попытается закрыть защищенный описатель, *CloseHandle* приведет к исключению. Необходимость в такой защите возникает очень редко. Однако этот флаг весьма полезен, когда процесс порождает дочерний, а тот в свою очередь — еще один процесс. При этом родительский процесс может ожидать, что его «внук» унаследует определенный описатель объекта, переданный дочернему. Но тут вполне возможно, что дочерний процесс, прежде чем породить новый процесс, закрывает нужный описатель. Тогда родительский процесс теряет связь с «внуком», поскольку тот не унаследовал требуемый объект ядра. Защитив описатель от закрытия, Вы исправите ситуацию, и «внук» унаследует предназначенный ему объект.

У этого подхода, впрочем, есть один недостаток. Дочерний процесс, вызвав:

```
SetHandleInformation(hobj, HANDLE_FLAG_PROTECT_FROM_CLOSE, 0);  
CloseHandle(hobj);
```

может сбросить флаг HANDLE_FLAG_PROTECT_FROM_CLOSE и закрыть затем соответствующий описатель. Родительский процесс ставит на то, что дочерний не исполнит этот код. Но одновременно он ставит и на то, что дочерний процесс породит ему «внука», поэтому в целом ставки не слишком рискованны.

Для полноты картины стоит, пожалуй, упомянуть и функцию *GetHandleInformation*:

```
BOOL GetHandleInformation(  
    HANDLE hObj,  
    PDWORD pdwFlags);
```

Эта функция возвращает текущие флаги для заданного описателя в переменной типа *DWORD*, на которую указывает *pdwFlags*. Чтобы проверить, является ли описатель наследуемым, сделайте так:

```
DWORD dwFlags;  
GetHandleInformation(hObj, &dwFlags);  
BOOL fHandleIsInheritable = (0 != (dwFlags & HANDLE_FLAG_INHERIT));
```

Именованные объекты

Второй способ, позволяющий нескольким процессам совместно использовать одни и те же объекты ядра, связан с именованием этих объектов. Именование допускают многие (но не все) объекты ядра. Например, следующие функции создают именованные объекты ядра:

```

HANDLE CreateMutex(
    PSECURITY_ATTRIBUTES psa,
    BOOL bInitialOwner,
    PCTSTR pszName);

HANDLE CreateEvent(
    PSECURITY_ATTRIBUTES psa,
    BOOL bManualReset,
    BOOL bInitialState,
    PCTSTR pszName);

HANDLE CreateSemaphore(
    PSECURITY_ATTRIBUTES psa,
    LONG lInitialCount,
    LONG lMaximumCount,
    PCTSTR pszName);

HANDLE CreateWaitableTimer(
    PSECURITY_ATTRIBUTES psa,
    BOOL bManualReset,
    PCTSTR pszName);

HANDLE CreateFileMapping(
    HANDLE hFile,
    PSECURITY_ATTRIBUTES psa,
    DWORD f1Protect,
    DWORD dwMaximumSizeHigh,
    DWORD dwMaximumSizeLow,
    PCTSTR pszName);

HANDLE CreateJobObject(
    PSECURITY_ATTRIBUTES psa,
    PCTSTR pszName);

```

Последний параметр, *pszName*, у всех этих функций одинаков. Передавая в нем NULL, Вы создаете безымянный (анонимный) объект ядра. В этом случае Вы можете разделять объект между процессами либо через наследование (см. предыдущий раздел), либо с помощью *DuplicateHandle* (см. следующий раздел). А чтобы разделять объект по имени, Вы должны присвоить ему какое-нибудь имя. Тогда вместо NULL в параметре *pszName* нужно передать адрес строки с именем, завершающей нулевым символом. Имя может быть длиной до MAX_PATH знаков (это значение определено как 260). К сожалению, Microsoft ничего не сообщает о правилах именования объектов ядра. Например, создавая объект с именем JeffObj, Вы никак не застрахованы от того, что в системе еще нет объекта ядра с таким именем. И что хуже, все эти объекты делят единное пространство имен. Из-за этого следующий вызов *CreateSemaphore* будет всегда возвращать NULL:

```

HANDLE hMutex = CreateMutex(NULL, FALSE, "JeffObj");
HANDLE hSem = CreateSemaphore(NULL, 1, 1, "JeffObj");
DWORD dwErrorCode = GetLastError();

```

После выполнения этого фрагмента значение *dwErrorCode* будет равно 6 (ERROR_INVALID_HANDLE). Полученный код ошибки не слишком вразумителен, но другого не дано.

Теперь, когда Вы научились именовать объекты, рассмотрим, как разделять их между процессами по именам. Допустим, после запуска процесса A вызывается функция:

```
HANDLE hMutexProcessA = CreateMutex(NULL, FALSE, "JeffMutex");
```

Этот вызов заставляет систему создать новенький, как с иголочки, объект ядра «мьютекс» и присвоить ему имя JeffMutex. Заметьте, что описатель *hMutexProcessA* в процессе A не является наследуемым, — он и не должен быть таковым при простом именовании объектов.

Спустя какое-то время некий процесс порождает процесс B. Необязательно, чтобы последний был дочерним от процесса A; он может быть порожден Explorer или любым другим приложением. (В этом, кстати, и состоит преимущество механизма именования объектов перед наследованием.) Когда процесс B приступает к работе, исполняется код:

```
HANDLE hMutexProcessB = CreateMutex(NULL, FALSE, "JeffMutex");
```

При этом вызове система сначала проверяет, не существует ли уже объект ядра с таким именем. Если да, то ядро проверяет тип этого объекта. Поскольку мы пытаемся создать мьютекс и его имя тоже JeffMutex, система проверяет права доступа вызывающего процесса к этому объекту. Если у него есть все права, в таблице описателей, принадлежащей процессу B, создается новая запись, указывающая на существующий объект ядра. Если же вызывающий процесс не имеет полных прав на доступ к объекту или если типы двух объектов с одинаковыми именами не совпадают, вызов *CreateMutex* заканчивается неудачно и возвращается NULL.

Однако, хотя процесс B успешно вызвал *CreateMutex*, новый объект-мьютекс он не создал. Вместо этого он получил свой описатель существующего объекта-мьютекса. Счетчик объекта, конечно же, увеличился на 1, и теперь этот объект не разрушится, пока его описатели не закроют оба процесса — A и B. Заметьте, что значения описателей объекта в обоих процессах скорее всего разные, но так и должно быть: каждый процесс будет оперировать с данным объектом ядра, используя свой описатель.



Разделяя объекты ядра по именам, помните об одной крайне важной вещи. Вызывая *CreateMutex*, процесс B передает ей атрибуты защиты и второй параметр. Так вот, эти параметры игнорируются, если объект с указанным именем уже существует! Приложение может определить, что оно делает: создает новый объект ядра или просто открывает уже существующий, — вызвав *GetLastError* сразу же после вызова одной из *Create*-функций:

```
HANDLE hMutex = CreateMutex(&sa, FALSE, "JeffObj");
if (GetLastError() == ERROR_ALREADY_EXISTS) {
    // открыт описатель существующего объекта;
    // sa.lpSecurityDescriptor и второй параметр
    // (FALSE) игнорируются
} else {
    // создан совершенно новый объект;
    // sa.lpSecurityDescriptor и второй параметр
    // (FALSE) используются при создании объекта
}
```

Есть и другой способ разделения объектов по именам. Вместо вызова *Create*-функции процесс может обратиться к одной из следующих *Open*-функций:

```

HANDLE OpenMutex(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);

HANDLE OpenEvent(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);

HANDLE OpenSemaphore(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);

HANDLE OpenWaitableTimer(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);

HANDLE OpenFileMapping(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);

HANDLE OpenJobObject(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);

```

Заметьте: все эти функции имеют один прототип. Последний параметр, *pszName*, определяет имя объекта ядра. В нем нельзя передать NULL — только адрес строки с нулевым символом в конце. Эти функции просматривают единое пространство имен объектов ядра, пытаясь найти совпадение. Если объекта ядра с указанным именем нет, функции возвращают NULL, а *GetLastError* — код 2 (ERROR_FILE_NOT_FOUND). Но если объект ядра с заданным именем существует и если его тип идентичен тому, что Вы указали, система проверяет, разрешен ли к данному объекту доступ запрошенного вида (через параметр *dwDesiredAccess*). Если такой вид доступа разрешен, таблица описателей в вызывающем процессе обновляется, и счетчик числа пользователей объекта возрастает на 1. Если Вы присвоили параметру *bInheritHandle* значение TRUE, то получите наследуемый описатель.

Главное отличие между вызовом *Create*- и *Open*-функций в том, что при отсутствии указанного объекта *Create*-функция создает его, а *Open*-функция просто уведомляет об ошибке.

Как я уже говорил, Microsoft ничего не сообщает о правилах именования объектов ядра. Но представьте себе, что пользователь запускает две программы от разных компаний и каждая программа пытается создать объект с именем «MyObject». Ничего хорошего из этого не выйдет. Чтобы избежать такой ситуации, я бы посоветовал со-здавать GUID и использовать его строковое представление как имя объекта.

Именованные объекты часто применяются для того, чтобы не допустить запуска нескольких экземпляров одного приложения. Для этого Вы просто вызываете одну из *Create*-функций в своей функции *main* или *WinMain* и создаете некий именованный

объект. Какой именно — не имеет ни малейшего значения. Сразу после *Create*-функции Вы должны вызвать *GetLastError*. Если она вернет ERROR_ALREADY_EXISTS, значит, один экземпляр Вашего приложения уже выполняется и новый его экземпляр можно закрыть. Вот фрагмент кода, иллюстрирующий этот прием:

```
int WINAPI WinMain(HINSTANCE hinstExe, HINSTANCE, PSTR pszCmdLine, int nCmdShow) {  
    HANDLE h = CreateMutex(NULL, FALSE,  
        "{FA531CC1-0497-11d3-A180-00105A276C3E}");  
    if (GetLastError() == ERROR_ALREADY_EXISTS) {  
        // экземпляр этого приложения уже выполняется  
        return(0);  
    }  
    // запущен первый экземпляр данного приложения  
  
    :  
  
    // перед выходом закрываем объект  
    CloseHandle(h);  
    return(0);  
}
```

Пространства имен Terminal Server

Terminal Server несколько меняет описанный выше сценарий. На машине с Terminal Server существует множество пространств имен для объектов ядра. Объекты, которые должны быть доступны всем клиентам, используют одно глобальное пространство имен. (Такие объекты, как правило, связаны с сервисами, предоставляемыми клиентским программам.) В каждом клиентском сеансе формируется свое пространство имен, чтобы исключить конфликты между несколькими сессиями, в которых запускается одно и то же приложение. Ни из какого сеанса нельзя получить доступ к объектам другого сеанса, даже если у их объектов идентичные имена.

Именованные объекты ядра, относящиеся к какому-либо сервису, всегда находятся в глобальном пространстве имен, а аналогичный объект, связанный с приложением, Terminal Server по умолчанию помещает в пространство имен клиентского сеанса. Однако и его можно перевести в глобальное пространство имен, поставив перед именем объекта префикс «Global\\», как в примере ниже.

```
HANDLE h = CreateEvent(NULL, FALSE, FALSE, "Global\\MyName");
```

Если Вы хотите явно указать, что объект ядра должен находиться в пространстве имен клиентского сеанса, используйте префикс «Local\\»:

```
HANDLE h = CreateEvent(NULL, FALSE, FALSE, "Local\\MyName");
```

Microsoft рассматривает префиксы Global и Local как зарезервированные ключевые слова, которые не должны встречаться в самих именах объектов. К числу таких слов Microsoft относит и Session, хотя на сегодняшний день оно не связано ни с какой функциональностью. Также обратите внимание на две вещи: все эти ключевые слова чувствительны к регистру букв и игнорируются, если компьютер работает без Terminal Server.

Дублирование описателей объектов

Последний механизм совместного использования объектов ядра несколькими процессами требует функции *DuplicateHandle*:

```
BOOL DuplicateHandle(
    HANDLE hSourceProcessHandle,
    HANDLE hSourceHandle,
    HANDLE hTargetProcessHandle,
    PHANDLE phTargetHandle,
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwOptions);
```

Говоря по-простому, эта функция берет запись в таблице описателей одного процесса и создает ее копию в таблице другого. *DuplicateHandle* принимает несколько параметров, но на самом деле весьма незамысловата. Обычно ее применение требует наличия в системе трех разных процессов.

Первый и третий параметры функции *DuplicateHandle* представляют собой описатели объектов ядра, специфичные для вызывающего процесса. Кроме того, эти параметры должны идентифицировать именно процессы — функция завершится с ошибкой, если Вы передадите описатели на объекты ядра любого другого типа. Подробнее объекты ядра «процессы» мы обсудим в главе 4, а сейчас Вам достаточно знать только одно: объект ядра «процесс» создается при каждой инициации в системе нового процесса.

Второй параметр, *hSourceHandle*, — описатель объекта ядра любого типа. Однако его значение специфично не для процесса, вызывающего *DuplicateHandle*, а для того, на который указывает описатель *hSourceProcessHandle*. Параметр *phTargetHandle* — это адрес переменной типа HANDLE, в которой возвращается индекс записи с копией описателя из процесса-источника. Значение возвращаемого описателя специфично для процесса, определяемого параметром *hTargetProcessHandle*.

Предпоследние два параметра *DuplicateHandle* позволяют задать маску доступа и флаг наследования, устанавливаемые для данного описателя в процессе-приемнике. И, наконец, параметр *dwOptions* может быть 0 или любой комбинацией двух флагов: DUPLICATE_SAME_ACCESS и DUPLICATE_CLOSE_SOURCE.

Первый флаг подсказывает *DuplicateHandle*: у описателя, получаемого процессом-приемником, должна быть та же маска доступа, что и у описателя в процессе-источнике. Этот флаг заставляет *DuplicateHandle* игнорировать параметр *dwDesiredAccess*.

Второй флаг приводит к закрытию описателя в процессе-источнике. Он позволяет процессам обмениваться объектом ядра как эстафетной палочкой. При этом счетчик объекта не меняется.

Попробуем проиллюстрировать работу функции *DuplicateHandle* на примере. Здесь S — это процесс-источник, имеющий доступ к какому-то объекту ядра, T — это процесс-приемник, который получит доступ к тому же объекту ядра, а C — процесс-катализатор, вызывающий функцию *DuplicateHandle*.

Таблица описателей в процессе С (см. таблицу 3-4) содержит два индекса — 1 и 2. Описатель с первым значением идентифицирует объект ядра «процесс S», описатель со вторым значением — объект ядра «процесс T».

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0xF0000000 (объект ядра процесса S)	0x????????	0x00000000
2	0xF0000010 (объект ядра процесса T)	0x????????	0x00000000

Таблица 3-4. Таблица описателей в процессе С

Таблица 3-5 иллюстрирует таблицу описателей в процессе S, содержащую единственную запись со значением описателя, равным 2. Этот описатель может идентифицировать объект ядра любого типа, а не только «процесс».

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0x00000000	(неприменим)	(неприменим)
2	0xF0000020 (объект ядра любого типа)	0x????????	0x00000000

Таблица 3-5. Таблица описателей в процессе S

В таблице 3-6 показано, что именно содержит таблица описателей в процессе T перед вызовом процессом С функции *DuplicateHandle*. Как видите, в ней всего одна запись со значением описателя, равным 2, а запись с индексом 1 пока пуста.

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0x00000000	(неприменим)	(неприменим)
2	0xF0000030 (объект ядра любого типа)	0x????????	0x00000000

Таблица 3-6. Таблица описателей в процессе T перед вызовом *DuplicateHandle*

Если процесс С теперь вызовет *DuplicateHandle* так:

```
DuplicateHandle(1, 2, 2, &hObj, 0, TRUE, DUPLICATE_SAME_ACCESS);
```

то после вызова изменится только таблица описателей в процессе T (см. таблицу 3-7).

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0xF0000020	0x????????	0x00000001
2	0xF0000030 (объект ядра любого типа)	0x????????	0x00000000

Таблица 3-7. Таблица описателей в процессе T после вызова *DuplicateHandle*

Вторая строка таблицы описателей в процессе S скопирована в первую строку таблицы описателей в процессе T. Функция *DuplicateHandle* присвоила также переменной *hObj* процесса С значение 1 — индекс той строки таблицы в процессе T, в которую занесен новый описатель.

Поскольку функции *DuplicateHandle* передан флаг DUPLICATE_SAME_ACCESS, маска доступа для этого описателя в процессе T идентична маске доступа в процессе S. Кроме того, данный флаг заставляет *DuplicateHandle* проигнорировать параметр *dwDesiredAccess*. Заметьте также, что система установила битовый флаг наследования, так как в параметре *bInheritHandle* функции *DuplicateHandle* мы передали TRUE.

Очевидно, Вы никогда не станете передавать в *DuplicateHandle* жестко зашитые значения, как это сделал я, просто демонстрируя работу функции. В реальных программах значения описателей хранятся в переменных и, конечно же, именно эти переменные передаются функциям.

Как и механизм наследования, функция *DuplicateHandle* тоже обладает одной странностью: процесс-приемник никак не уведомляется о том, что он получил доступ к новому объекту ядра. Поэтому процесс С должен каким-то образом сообщить

процессу Т, что тот имеет теперь доступ к новому объекту; для этого нужно воспользоваться одной из форм межпроцессной связи и передать в процесс Т значение описателя в переменной *hObj*. Ясное дело, в данном случае не годится ни командная строка, ни изменение переменных окружения процесса Т, поскольку этот процесс уже выполняется. Здесь придется послать сообщение окну или задействовать какой-нибудь другой механизм межпроцессной связи.

Я рассказал Вам о функции *DuplicateHandle* в самом общем виде. Надеюсь, Вы увидели, насколько она гибка. Но эта функция редко используется в ситуациях, требующих участия трех разных процессов. Обычно ее вызывают применительно к двум процессам. Представьте, что один процесс имеет доступ к объекту, к которому хочет обратиться другой процесс, или что один процесс хочет предоставить другому доступ к «своему» объекту ядра. Например, если процесс S имеет доступ к объекту ядра и Вам нужно, чтобы к этому объекту мог обращаться процесс Т, используйте *DuplicateHandle* так:

```
// весь приведенный ниже код исполняется процессом S

// создаем объект-мьютекс, доступный процессу S
HANDLE hObjProcessS = CreateMutex(NULL, FALSE, NULL);

// открываем описатель объекта ядра "процесс Т"
HANDLE hProcessT = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwProcessIdT);

HANDLE hObjProcessT; // неинициализированный описатель,
// связанный с процессом Т

// предоставляем процессу Т доступ к объекту-мьютексу
DuplicateHandle(GetCurrentProcess(), hObjProcessS, hProcessT,
&hObjProcessT, 0, FALSE, DUPLICATE_SAME_ACCESS);

// используем какую-нибудь форму межпроцессной связи, чтобы передать
// значение описателя из hObjProcessS в процесс Т

:

// связь с процессом Т больше не нужна
CloseHandle(hProcessT);

:

// если процессу S не нужен объект-мьютекс, он должен закрыть его
CloseHandle(hObjProcessS);
```

Вызов *GetCurrentProcess* возвращает псевдоописатель, который всегда идентифицирует вызывающий процесс, в данном случае — процесс S. Как только функция *DuplicateHandle* возвращает управление, *hObjProcessT* становится описателем, связанным с процессом Т и идентифицирующим тот же объект, что и описатель *hObjProcessS* (когда на него ссылается код процесса S). При этом процесс S ни в коем случае не должен исполнять следующий код:

```
// процесс S никогда не должен пытаться исполнять код,
// закрывающий продублированный описатель
CloseHandle(hObjProcessT);
```

Если процесс S выполнит этот код, вызов может дать (а может и не дать) ошибку. Он будет успешен, если у процесса S случайно окажется описатель с тем же значением, что и в *hObjProcessT*. При этом процесс S закроет неизвестно какой объект, и что будет потом — остается только гадать.

Теперь о другом способе применения *DuplicateHandle*. Допустим, некий процесс имеет полный доступ (для чтения и записи) к объекту «проекция файла» и из этого процесса вызывается функция, которая должна обратиться к проекции файла и считать из нее какие-то данные. Так вот, если мы хотим повысить отказоустойчивость приложения, то могли бы с помощью *DuplicateHandle* создать новый описатель существующего объекта и разрешить доступ только для чтения. Потом мы передали бы этот описатель функции, и та уже не смогла бы случайно что-то записать в проекцию файла. Взгляните на код, который иллюстрирует этот пример:

```
int WINAPI WinMain(HINSTANCE hinstExe, HINSTANCE,
    PSTR pszCmdLine, int nCmdShow) {

    // создаем объект "проекция файла";
    // его описатель разрешает доступ как для чтения, так и для записи
    HANDLE hFileMapRW = CreateFileMapping(INVALID_HANDLE_VALUE,
        NULL, PAGE_READWRITE, 0, 10240, NULL);

    // создаем другой описатель на тот же объект;
    // этот описатель разрешает доступ только для чтения
    HANDLE hFileMapRO;
    DuplicateHandle(GetCurrentProcess(), hFileMapRW, GetCurrentProcess(),
        &hFileMapRO, FILE_MAP_READ, FALSE, 0);

    // вызываем функцию, которая не должна ничего записывать в проекцию файла
    ReadFromTheFileMapping(hFileMapRO);

    // закрываем объект "проекция файла", доступный только для чтения
    CloseHandle(hFileMapRO);

    // проекция файла нам по-прежнему полностью доступна через hFileMapRW
    :

    // если проекция файла больше не нужна основному коду, закрываем ее
    CloseHandle(hFileMapRW);
}
```

ЧАСТЬ II

НАЧИНАЕМ РАБОТАТЬ



Процессы

Эта глава о том, как система управляет выполняемыми приложениями. Сначала я определию понятие «процесс» и объясню, как система создает объект ядра «процесс». Затем я покажу, как управлять процессом, используя сопоставленный с ним объект ядра. Далее мы обсудим атрибуты (или свойства) процесса и поговорим о нескольких функциях, позволяющих обращаться к этим свойствам и изменять их. Я расскажу также о функциях, которые создают (порождают) в системе дополнительные процессы. Ну и, конечно, описание процессов было бы неполным, если бы я не рассмотрел механизм их завершения. О'кэй, приступим.

Процесс обычно определяют как экземпляр выполняемой программы, и он состоит из двух компонентов:

- объекта ядра, через который операционная система управляет процессом. Там же хранится статистическая информация о процессе;
- адресного пространства, в котором содержится код и данные всех EXE- и DLL-модулей. Именно в нем находятся области памяти, динамически распределяемой для стеков потоков и других нужд.

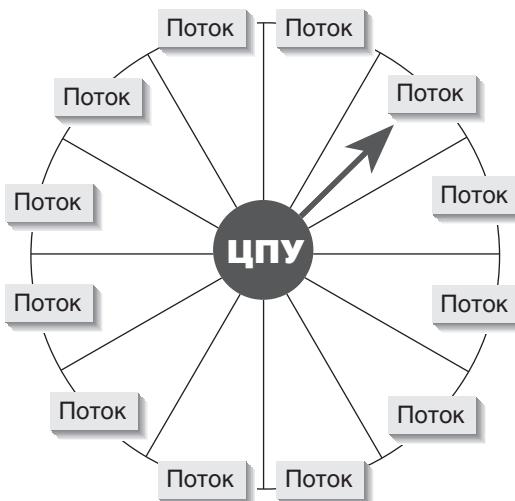


Рис. 4-1. Операционная система выделяет потокам кванты времени по принципу карусели

Процессы инертны. Чтобы процесс что-нибудь выполнил, в нем нужно создать поток. Именно потоки отвечают за исполнение кода, содержащегося в адресном пространстве процесса. В принципе, один процесс может владеть несколькими потоками, и тогда они «одновременно» исполняют код в адресном пространстве процесса.

Для этого каждый поток должен располагать собственным набором регистров процессора и собственным стеком. В каждом процессе есть минимум один поток. Если бы у процесса не было ни одного потока, ему нечего было бы делать на этом свете, и система автоматически уничтожила бы его вместе с выделенным ему адресным пространством.

Чтобы все эти потоки работали, операционная система отводит каждому из них определенное процессорное время. Выделяя потокам отрезки времени (называемые *квантами*) по принципу карусели, она создает тем самым иллюзию одновременного выполнения потоков. Рис. 4-1 иллюстрирует распределение процессорного времени между потоками на машине с одним процессором. Если в машине установлено более одного процессора, алгоритм работы операционной системы значительно усложняется (в этом случае система стремится сбалансировать нагрузку между процессорами).

При создании процесса первый (точнее, первичный) поток создается системой автоматически. Далее этот поток может породить другие потоки, те в свою очередь — новые и т. д.

WINDOWS 2000 Windows 2000 в полной мере использует возможности машин с несколькими процессорами. Например, эту книгу я писал, сидя за машиной с двумя процессорами. Windows 2000 способна закрепить каждый поток за отдельным процессором, и тогда два потока исполняются действительно одновременно. Ядро Windows 2000 полностью поддерживает распределение процессорного времени между потоками и управление ими на таких системах. Вам не придется делать ничего особенного в своем коде, чтобы задействовать преимущества многопроцессорной машины.

WINDOWS 98 Windows 98 работает только с одним процессором. Даже если у компьютера несколько процессоров, под управлением Windows 98 действует лишь один из них — остальные пристаивают.

Ваше первое Windows-приложение

Windows поддерживает два типа приложений: основанные на графическом интерфейсе (graphical user interface, GUI) и консольные (console user interface, CUI). У приложений первого типа внешний интерфейс чисто графический. GUI-приложения создают окна, имеют меню, взаимодействуют с пользователем через диалоговые окна и вообще пользуются всей стандартной «Windows'овской» начинкой. Почти все стандартные программы Windows — Notepad, Calculator, Wordpad и др. — являются GUI-приложениями. Приложения консольного типа работают в текстовом режиме: они не формируют окна, не обрабатывают сообщения и не требуют GUI. И хотя консольные приложения на экране тоже размещаются в окне, в нем содержится только текст. Командные процессоры вроде Cmd.exe (в Windows 2000) или Command.com (в Windows 98) — типичные образцы подобных приложений.

Вместе с тем граница между двумя типами приложений весьма условна. Можно, например, создать консольное приложение, способное отображать диалоговые окна. Скажем, в командном процессоре вполне может быть специальная команда, открывающая графическое диалоговое окно со списком команд; вроде мелочь — а избавляет от запоминания лишней информации. В то же время можно создать и GUI-приложение, выводящее текстовые строки в консольное окно. Я сам часто писал такие про-

грамммы: создав консольное окно, я пересыпал в него отладочную информацию, связанную с исполняемым приложением. Но, конечно, графический интерфейс предпочтительнее, чем старомодный текстовый. Как показывает опыт, приложения на основе GUI «дружественнее» к пользователю, а значит и более популярны.

Когда Вы создаете проект приложения, Microsoft Visual C++ устанавливает такие ключи для компоновщика, чтобы в исполняемом файле был указан соответствующий тип подсистемы. Для CUI-программ используется ключ /SUBSYSTEM:CONSOLE, а для GUI-приложений — /SUBSYSTEM:WINDOWS. Когда пользователь запускает приложение, загрузчик операционной системы проверяет номер подсистемы, хранящийся в заголовке образа исполняемого файла, и определяет, что это за программа — GUI или CUI. Если номер указывает на приложение последнего типа, загрузчик автоматически создает текстовое консольное окно, а если номер свидетельствует о противоположном — просто загружает программу в память. После того как приложение начинает работать, операционная система больше не интересуется, к какому типу оно относится.

Во всех Windows-приложениях должна быть входная функция, за реализацию которой отвечаете Вы. Существует четыре такие функции:

```
int WINAPI WinMain(
    HINSTANCE hinstExe,
    HINSTANCE,
    PSTR pszCmdLine,
    int nCmdShow);

int WINAPI wWinMain(
    HINSTANCE hinstExe,
    HINSTANCE,
    PWSTR pszCmdLine,
    int nCmdShow);

int __cdecl main(
    int argc,
    char *argv[],
    char *envp[]);

int __cdecl wmain(
    int argc,
    wchar_t *argv[],
    wchar_t *envp[]);
```

На самом деле входная функция операционной системой не вызывается. Вместо этого происходит обращение к стартовой функции из библиотеки C/C++. Она инициализирует библиотеку C/C++, чтобы можно было вызывать такие функции, как *malloc* и *free*, а также обеспечивает корректное создание любых объявленных Вами глобальных и статических C++-объектов до того, как начнется выполнение Вашего кода. В следующей таблице показано, в каких случаях реализуются те или иные входные функции.

Тип приложения	Входная функция	Стартовая функция, встраиваемая в Ваш исполняемый файл
GUI-приложение, работающее с ANSI-символами и строками	<i>WinMain</i>	<i>WinMainCRTStartup</i>
GUI-приложение, работающее с Unicode-символами и строками	<i>wWinMain</i>	<i>wWinMainCRTStartup</i>
CUI-приложение, работающее с ANSI-символами и строками	<i>main</i>	<i>mainCRTStartup</i>
CUI-приложение, работающее с Unicode-символами и строками	<i>wmain</i>	<i>wmainCRTStartup</i>

Нужную стартовую функцию в библиотеке C/C++ выбирает компоновщик при сборке исполняемого файла. Если указан ключ /SUBSYSTEM:WINDOWS, компоновщик ищет в Вашем коде функцию *WinMain* или *wWinMain*. Если ни одной из них нет, он сообщает об ошибке «unresolved external symbol» («неразрешенный внешний символ»); в ином случае — выбирает *WinMainCRTStartup* или *wWinMainCRTStartup* соответственно.

Аналогичным образом, если задан ключ /SUBSYSTEM:CONSOLE, компоновщик ищет в коде функцию *main* или *wmain* и выбирает соответственно *mainCRTStartup* или *wmainCRTStartup*; если в коде нет ни *main*, ни *wmain*, сообщается о той же ошибке — «unresolved external symbol».

Но не многие знают, что в проекте можно вообще не указывать ключ /SUBSYSTEM компоновщика. Если Вы так и сделаете, компоновщик будет сам определять подсистему для Вашего приложения. При компоновке он проверит, какая из четырех функций (*WinMain*, *wWinMain*, *main* или *wmain*) присутствует в Вашем коде, и на основании этого выберет подсистему и стартовую функцию из библиотеки C/C++.

Одна из частых ошибок, допускаемых теми, кто лишь начинает работать с Visual C++, — выбор неверного типа проекта. Например, разработчик хочет создать проект Win32 Application, а сам включает в код функцию *main*. При его сборке он получает сообщение об ошибке, так как для проекта Win32 Application в командной строке компоновщика автоматически указывается ключ /SUBSYSTEM:WINDOWS, который требует присутствия в коде функции *WinMain* или *wWinMain*. В этот момент разработчик может выбрать один из четырех вариантов дальнейших действий:

- заменить *main* на *WinMain*. Как правило, это не лучший вариант, поскольку разработчик скорее всего и хотел создать консольное приложение;
- открыть новый проект, на этот раз — Win32 Console Application, и перенести в него все модули кода. Этот вариант весьма утомителен, и возникает ощущение, будто начинаешь все заново;
- открыть вкладку Link в диалоговом окне Project Settings и заменить ключ /SUBSYSTEM:WINDOWS на /SUBSYSTEM:CONSOLE. Некоторые думают, что это единственный вариант;
- открыть вкладку Link в диалоговом окне Project Settings и вообще убрать ключ /SUBSYSTEM:WINDOWS. Я предпочитаю именно этот способ, потому что он самый гибкий. Компоновщик сам сделает все, что надо, в зависимости от входной функции, которую Вы реализуете в своем коде. Никак не пойму, почему это не предлагается по умолчанию при создании нового проекта Win32 Application или Win32 Console Application.

Все стартовые функции из библиотеки C/C++ делают практически одно и то же. Разница лишь в том, какие строки они обрабатывают (в ANSI или Unicode) и какую входную функцию вызывают после инициализации библиотеки. Кстати, с Visual C++ поставляется исходный код этой библиотеки, и стартовые функции находятся в файле CRt0.c. А теперь рассмотрим, какие операции они выполняют:

- считывают указатель на полную командную строку нового процесса;
- считывают указатель на переменные окружения нового процесса;
- инициализируют глобальные переменные из библиотеки C/C++, доступ к которым из Вашего кода обеспечивается включением файла StdLib.h. Список этих переменных приведен в таблице 4-1;
- инициализируют кучу (динамически распределяемую область памяти), используемую C-функциями выделения памяти (т. е. *malloc* и *calloc*) и другими процедурами низкоуровневого ввода-вывода;
- вызывают конструкторы всех глобальных и статических объектов C++-классов.

Закончив эти операции, стартовая функция обращается к входной функции в Вашей программе. Если Вы написали ее в виде *wWinMain*, то она вызывается так:

```
GetStartupInfo(&StartupInfo);
int nMainRetVal = wWinMain(GetModuleHandle(NULL), NULL, pszCommandLineUnicode,
(StartupInfo.dwFlags & STARTF_USESHOWWINDOW)
? StartupInfo.wShowWindow : SW_SHOWDEFAULT);
```

А если Вы предпочли *WinMain*, то:

```
GetStartupInfo(&StartupInfo);
int nMainRetVal = WinMain(GetModuleHandle(NULL), NULL, pszCommandLineANSI,
(StartupInfo.dwFlags & STARTF_USESHOWWINDOW)
? StartupInfo.wShowWindow : SW_SHOWDEFAULT);
```

И, наконец, то же самое для функций *wmain* и *main*:

```
int nMainRetVal = wmain(__argc, __wargv, __wenviron);
int nMainRetVal = main(__argc, __argv, __environ);
```

Когда Ваша входная функция возвращает управление, стартовая обращается к функции *exit* библиотеки C/C++ и передает ей значение *nMainRetVal*. Функция *exit* выполняет следующие операции:

- вызывает все функции, зарегистрированные вызовами функции *_onexit*;
- вызывает деструкторы всех глобальных и статических объектов C++-классов;
- вызывает Windows-функцию *ExitProcess*, передавая ей значение *nMainRetVal*. Это заставляет операционную систему уничтожить Ваш процесс и установить код его завершения.

Имя переменной	Тип	Описание
<i>_osver</i>	<i>unsigned int</i>	Версия сборки операционной системы. Например, у Windows 2000 Beta 3 этот номер был 2031, соответственно <i>_osver</i> равна 2031.
<i>_winmajor</i>	<i>unsigned int</i>	Основной номер версии Windows в шестнадцатеричной форме. Для Windows 2000 это значение равно 5.

Таблица 4-1. Глобальные переменные из библиотеки C/C++, доступные Вашим программам

продолжение

Имя переменной	Тип	Описание
<code>_winminor</code>	<code>unsigned int</code>	Дополнительный номер версии Windows в шестнадцатеричной форме. Для Windows 2000 это значение равно 0.
<code>_winver</code>	<code>unsigned int</code>	Вычисляется как (<code>_winmajor << 8</code>) + <code>_winminor</code> .
<code>_argc</code>	<code>unsigned int</code>	Количество аргументов, переданных в командной строке.
<code>_argv</code> <code>_wargv</code>	<code>char **</code> <code>wchar_t **</code>	Массив размером <code>_argc</code> с указателями на ANSI- или Unicode-строки. Каждый элемент массива указывает на один из аргументов командной строки.
<code>_environ</code> <code>_wenviron</code>	<code>char **</code> <code>wchar_t **</code>	Массив указателей на ANSI- или Unicode-строки. Каждый элемент массива указывает на строку — переменную окружения.
<code>_pgmptr</code> <code>_wpgmptr</code>	<code>char **</code> <code>wchar_t **</code>	Полный путь и имя (в ANSI или Unicode) запускаемой программы.

Описатель экземпляра процесса

Любому EXE- или DLL-модулю, загружаемому в адресное пространство процесса, присваивается уникальный описатель экземпляра. Описатель экземпляра Вашего EXE-файла передается как первый параметр функции (*w*)*WinMain* — *hinstExe*. Это значение обычно требуется при вызовах функций, загружающих те или иные ресурсы. Например, чтобы загрузить из образа EXE-файла такой ресурс, как значок, надо вызвать:

```
HICON LoadIcon(
    HINSTANCE hinst,
    PCTSTR pszIcon);
```

Первый параметр в *LoadIcon* указывает, в каком файле (EXE или DLL) содержится интересующий Вас ресурс. Многие приложения сохраняют параметр *hinstExe* функции (*w*)*WinMain* в глобальной переменной, благодаря чему он доступен из любой части кода EXE-файла.

В документации Platform SDK утверждается, что некоторые Windows-функции требуют параметр типа `HMODULE`. Пример — функция *GetModuleFileName*:

```
DWORD GetModuleFileName(
    HMODULE hinstModule,
    PTSTR pszPath,
    DWORD cchPath);
```



Как оказалось, `HMODULE` и `HINSTANCE` — это одно и то же. Встретив в документации указание передать какой-то функции `HMODULE`, смело передавайте `HINSTANCE`, и наоборот. Они существуют в таком виде лишь потому, что в 16-разрядной Windows идентифицировали совершенно разные вещи.

Истинное значение параметра *hinstExe* функции (*w*)*WinMain* — базовый адрес в памяти, определяющий ту область в адресном пространстве процесса, куда был загружен образ данного EXE-файла. Например, если система открывает исполняемый файл и загружает его содержимое по адресу 0x00400000, то *hinstExe* функции (*w*)*WinMain* получает значение 0x00400000.

Базовый адрес, по которому загружается приложение, определяется компоновщиком. Разные компоновщики выбирают и разные (по умолчанию) базовые адреса. Компоновщик Visual C++ использует по умолчанию базовый адрес 0x00400000 — самый нижний в Windows 98, начиная с которого в ней допускается загрузка образа исполняемого файла. Указав параметр /BASE: *адрес* (в случае компоновщика от Microsoft), можно изменить базовый адрес, по которому будет загружаться приложение.

При попытке загрузить исполняемый файл в Windows 98 по базовому адресу ниже 0x00400000 загрузчик переместит его на другой адрес. Это увеличит время загрузки приложения, но оно по крайней мере будет выполнено. Если Вы разрабатываете программы и для Windows 98, и для Windows 2000, сделайте так, чтобы приложение загружалось по базовому адресу не ниже 0x00400000.

Функция *GetModuleHandle*:

```
MODULE GetModuleHandle(  
    PCTSTR pszModule);
```

возвращает описатель/базовый адрес, указывающий, куда именно (в адресном пространстве процесса) загружается EXE- или DLL-файл. При вызове этой функции имя нужного EXE- или DLL-файла передается как строка с нулевым символом в конце. Если система находит указанный файл, *GetModuleHandle* возвращает базовый адрес, по которому располагается образ данного файла. Если же файл системой не найден, функция возвращает NULL. Кроме того, можно вызвать эту функцию, передав ей NULL вместо параметра *pszModule*, — тогда Вы узнаете базовый адрес EXE-файла. Именно это и делает стартовый код из библиотеки C/C++ при вызове (*w*)*WinMain* из Вашей программы.

Есть еще две важные вещи, касающиеся *GetModuleHandle*. Во-первых, она проверяет адресное пространство только того процесса, который ее вызвал. Если этот процесс не использует никаких функций, связанных со стандартными диалоговыми окнами, то, вызвав *GetModuleHandle* и передав ей аргумент «ComDlg32», Вы получите NULL — пусть даже модуль ComDlg32.dll и загружен в адресное пространство какого-нибудь другого процесса. Во-вторых, вызов этой функции и передача ей NULL дает в результате базовый адрес EXE-файла в адресном пространстве процесса. Так что, вызывая функцию в виде *GetModuleHandle(NULL)* — даже из кода в DLL, — Вы получаете базовый адрес EXE-, а не DLL-файла.

Описатель предыдущего экземпляра процесса

Я уже говорил, что стартовый код из библиотеки C/C++ всегда передает в функцию (*w*)*WinMain* параметр *hinstExePrev* как NULL. Этот параметр предусмотрен исключительно для совместимости с 16-разрядными версиями Windows и не имеет никакого смысла для Windows-приложений. Поэтому я всегда пишу заголовок (*w*)*WinMain* так:

```
int WINAPI WinMain(  
    HINSTANCE hinstExe,  
    HINSTANCE,  
    PSTR pszCmdLine,  
    int nCmdShow);
```

Поскольку у второго параметра нет имени, компилятор не выдает предупреждение «parameter not referenced» («нет ссылки на параметр»).

Командная строка процесса

При создании новому процессу передается командная строка, которая почти никогда не бывает пустой — как минимум, она содержит имя исполняемого файла, использованного при создании этого процесса. Однако, как Вы увидите ниже (при обсуждении функции *CreateProcess*), возможны случаи, когда процесс получает командную строку, состоящую из единственного символа — нуля, завершающего строку. В момент запуска приложения стартовый код из библиотеки C/C++ считывает командную строку процесса, пропускает имя исполняемого файла и заносит в параметр *pszCmdLine* функции (*w*)*WinMain* указатель на оставшуюся часть командной строки.

Параметр *pszCmdLine* всегда указывает на ANSI-строку. Но, заменив *WinMain* на *wWinMain*, Вы получите доступ к Unicode-версии командной строки для своего процесса.

Программа может анализировать и интерпретировать командную строку как угодно. Поскольку *pszCmdLine* относится к типу PSTR, а не PCSTR, не стесняйтесь и записывайте строку прямо в буфер, на который указывает этот параметр, но ни при каких условиях не переступайте границу буфера. Лично я всегда рассматриваю этот буфер как «только для чтения». Если в командную строку нужно внести изменения, я сначала копирую буфер, содержащий командную строку, в локальный буфер (в своей программе), который затем и модифицирую.

Указатель на полную командную строку процесса можно получить и вызовом функции *GetCommandLine*:

```
PTSTR GetCommandLine();
```

Она возвращает указатель на буфер, содержащий полную командную строку, включая полное имя (вместе с путем) исполняемого файла.

Во многих приложениях безусловно удобнее использовать командную строку, предварительно разбитую на отдельные компоненты, доступ к которым приложение может получить через глобальные переменные *_argc* и *_argv* (или *_wargv*). Функция *CommandLineToArgvW* расщепляет Unicode-строку на отдельные компоненты:

```
PWSTR CommandLineToArgvW(
    PWSTR pszCmdLine,
    int pNumArgs);
```

Буква *W* в конце имени этой функции намекает на «широкие» (wide) символы и подсказывает, что функция существует только в Unicode-версии. Параметр *pszCmdLine* указывает на командную строку. Его обычно получают предварительным вызовом *GetCommandLineW*. Параметр *pNumArgs* — это адрес целочисленной переменной, в которой задается количество аргументов в командной строке. Функция *CommandLineToArgvW* возвращает адрес массива указателей на Unicode-строки.

CommandLineToArgvW выделяет нужную память автоматически. Большинство приложений не освобождает эту память, полагаясь на операционную систему, которая проводит очистку ресурсов по завершении процесса. И такой подход вполне приемлем. Но если Вы хотите сами освободить эту память, сделайте так:

```
int pNumArgs;
PWSTR *ppArgv = CommandLineToArgvW(GetCommandLineW(), &pNumArgs);
```

см. след. стр.

```
// используйте эти аргументы...
if (*ppArgv[1] == L'x') {
    :
    // освободите блок памяти
    HeapFree(GetProcessHeap(), 0, ppArgv);
```

Переменные окружения

С любым процессом связан блок переменных окружения — область памяти, выделенная в адресном пространстве процесса. Каждый блок содержит группу строк такого вида:

```
VarName1=VarValue1\0
VarName2=VarValue2\0
VarName3=VarValue3\0
:
VarNameX=VarValueX\0
\0
```

Первая часть каждой строки — имя переменной окружения. За ним следует знак равенства и значение, присваиваемое переменной. Строки в блоке переменных окружения должны быть отсортированы в алфавитном порядке по именам переменных.

Знак равенства разделяет имя переменной и ее значение, так что его нельзя использовать как символ в имени переменной. Важную роль играют и пробелы. Например, объявив две переменные:

```
XYZ= Windows      ( обратите внимание на пробел за знаком равенства)
ABC=Windows
```

и сравнив значения переменных *XYZ* и *ABC*, Вы увидите, что система их различает, — она учитывает любой пробел, поставленный перед знаком равенства или после него. Вот что будет, если записать, скажем, так:

```
XYZ =Home        ( обратите внимание на пробел перед знаком равенства)
XYZ=Work
```

Вы получите первую переменную с именем «*XYZ*», содержащую строку «*Home*», и вторую переменную «*XYZ*», содержащую строку «*Work*».

Конец блока переменных окружения помечается дополнительным нулевым символом.

WINDOWS 98 Чтобы создать исходный набор переменных окружения для Windows 98, надо модифицировать файл Autoexec.bat, поместив в него группу строк SET в виде:
SET VarName=VarValue

При перезагрузке система учтет новое содержимое файла Autoexec.bat, и тогда любые заданные Вами переменные окружения станут доступны всем процессам, инициируемым в сеансе работы с Windows 98.

WINDOWS 2000 При регистрации пользователя на входе в Windows 2000 система создает процесс-оболочку, связывая с ним группу строк — переменных окружения. Система получает начальные значения этих строк, анализируя два раздела в реестре. В первом:

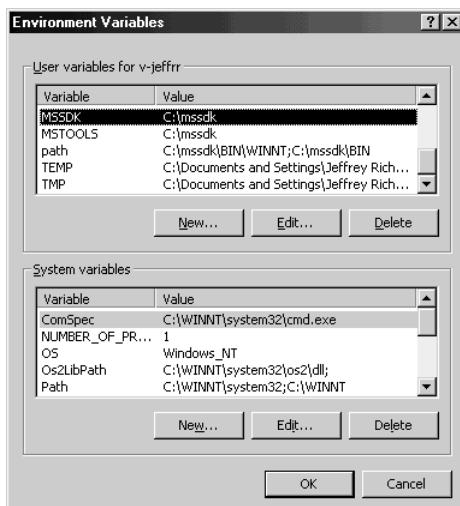
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SessionManager\Environment

содержится список переменных окружения, относящихся к системе, а во втором:

HKEY_CURRENT_USER\Environment

находится список переменных окружения, относящихся к пользователю, который в настоящее время зарегистрирован в системе.

Пользователь может добавлять, удалять или изменять любые переменные через аплет System из Control Panel. В этом аплете надо открыть вкладку Advanced и щелкнуть кнопку Environment Variables — тогда на экране появится следующее диалоговое окно.



Модифицировать переменные из списка System Variables разрешается только пользователю с правами администратора.

Кроме того, для модификации записей в реестре Ваша программа может обращаться к Windows-функциям, позволяющим манипулировать с реестром. Однако, чтобы изменения вступили в силу, пользователь должен выйти из системы и вновь войти в нее. Некоторые приложения типа Explorer, Task Manager или Control Panel могут обновлять свои блоки переменных окружения на базе новых значений в реестре, когда их главные окна получают сообщение WM_SETTINGCHANGE. Например, если Вы, изменив реестр, хотите, чтобы какие-то приложения соответственно обновили свои блоки переменных окружения, вызовите:

```
SendMessage(HWND_BROADCAST, WM_SETTINGCHANGE,
0, (LPARAM) TEXT("Environment"));
```

Обычно дочерний процесс наследует набор переменных окружения от родительского. Однако последний способен управлять тем, какие переменные окружения наследуются дочерним процессом, а какие — нет. Но об этом я расскажу, когда мы най-

мемся функцией *CreateProcess*. Под наследованием я имею в виду, что дочерний процесс получает свою копию блока переменных окружения от родительского, а не то, что дочерний и родительский процессы совместно используют один и тот же блок. Так что дочерний процесс может добавлять, удалять или модифицировать переменные в своем блоке, и эти изменения не затронут блок, принадлежащий родительскому процессу.

Переменные окружения обычно применяются для тонкой настройки приложения. Пользователь создает и инициализирует переменную окружения, затем запускает приложение, и оно, обнаружив эту переменную, проверяет ее значение и соответствующим образом настраивается.

Увы, многим пользователям не под силу разобраться в переменных окружения, а значит, трудно указать правильные значения. Ведь для этого надо не только хорошо знать синтаксис переменных, но и, конечно, понимать, что стоит за теми или иными их значениями. С другой стороны, почти все (а может, и все) приложения, основанные на GUI, дают возможность тонкой настройки через диалоговые окна. Такой подход, естественно, нагляднее и проще.

А теперь, если у Вас еще не пропало желание манипулировать переменными окружения, поговорим о предназначенных для этой цели функциях. *GetEnvironmentVariable* позволяет выявлять присутствие той или иной переменной окружения и определять ее значение:

```
DWORD GetEnvironmentVariable(
    PCTSTR pszName,
    PTSTR pszValue,
    DWORD cchValue);
```

При вызове *GetEnvironmentVariable* параметр *pszName* должен указывать на имя интересующей Вас переменной, *pszValue* — на буфер, в который будет помещено значение переменной, а в *cchValue* следует сообщить размер буфера в символах. Функция возвращает либо количество символов, скопированных в буфер, либо 0, если ей удалось обнаружить переменную окружения с таким именем.

Кстати, в реестре многие строки содержат подставляемые части, например:

```
%USERPROFILE%\My Documents
```

Часть, заключенная в знаки процента, является подставляемой. В данном случае в строку должно быть подставлено значение переменной окружения *USERPROFILE*. На моей машине эта переменная выглядит так:

```
C:\Documents and Settings\Administrator
```

После подстановки переменной в строку реестра получим:

```
C:\Documents and Settings\Administrator\My Documents
```

Поскольку такие подстановки делаются очень часто, в Windows есть функция *ExpandEnvironmentStrings*:

```
DWORD ExpandEnvironmentStrings(
    PCTSTR pszSrc,
    PTSTR pszDst,
    DWORD nSize);
```

Параметр *pszSrc* принимает адрес строки, содержащей подставляемые части, а параметр *pszDst* — адрес буфера, в который записывается развернутая строка. Параметр *nSize* определяет максимальный размер буфера в символах.

Наконец, функция *SetEnvironmentVariable* позволяет добавлять, удалять и модифицировать значение переменной:

```
DWORD SetEnvironmentVariable(
    PCTSTR pszName,
    PCTSTR pszValue);
```

Она устанавливает ту переменную, на чье имя указывает параметр *pszName*, и присваивает ей значение, заданное параметром *pszValue*. Если такая переменная уже существует, функция модифицирует ее значение. Если же в *pszValue* содержится NULL, переменная удаляется из блока.

Для манипуляций с блоком переменных окружения всегда используйте именно эти функции. Как я уже говорил, строки в блоке переменных нужно отсортировать в алфавитном порядке по именам переменных (тогда *GetEnvironmentVariable* быстрее находит нужные переменные), а *SetEnvironmentVariable* как раз и следит за порядком расположения переменных.

Привязка к процессорам

Обычно потоки внутри процесса могут выполняться на любом процессоре компьютера. Однако их можно закрепить за определенным подмножеством процессоров из числа имеющихся на компьютере. Это свойство называется *привязкой к процессорам* (processor affinity) и подробно обсуждается в главе 7. Дочерние процессы наследуют привязку к процессорам от родительских.

Режим обработки ошибок

С каждым процессом связан набор флагов, сообщающих системе, каким образом процесс должен реагировать на серьезные ошибки: повреждения дисковых носителей, необрабатываемые исключения, ошибки операций поиска файлов и неверное выравнивание данных. Процесс может указать системе, как обрабатывать каждую из этих ошибок, через функцию *SetErrorMode*:

```
UINT SetErrorMode(UINT fuErrorMode);
```

Параметр *fuErrorMode* — это набор флагов, комбинируемых побитовой операцией OR:

Флаг	Описание
SEM_FAILCRITICALERRORS	Система не выводит окно с сообщением от обработчика критических ошибок и возвращает ошибку в вызывающий процесс
SEM_NOGPFAULTERRORBOX	Система не выводит окно с сообщением о нарушении общей защиты; этим флагом манипулируют только отладчики, самостоятельно обрабатывающие нарушения общей защиты с помощью обработчика исключений
SEM_NOOPENFILEERRORBOX	Система не выводит окно с сообщением об отсутствии искомого файла
SEM_NOALIGNMENTFAULTEXCEPT	Система автоматически исправляет нарушения в выравнивании данных, и они становятся невидимы приложению; этот флаг не действует на процессорах x86

По умолчанию дочерний процесс наследует от родительского флаги, указывающие на режим обработки ошибок. Иначе говоря, если у процесса в данный момент установлен флаг SEM_NOGPFAULTERRORBOX и он порождает другой процесс, этот

флаг будет установлен и у дочернего процесса. Однако «наследник» об этом не уведомляется, и он вообще может быть не рассчитан на обработку ошибок такого типа (в данном случае — нарушений общей защиты). В результате, если в одном из потоков дочернего процесса все-таки произойдет подобная ошибка, этот процесс может завершиться, ничего не сообщив пользователю. Но родительский процесс способен предотвратить наследование дочерним процессом своего режима обработки ошибок, указав при вызове функции *CreateProcess* флаг *CREATE_DEFAULT_ERROR_MODE* (о *CreateProcess* чуть позже).

Текущие диск и каталог для процесса

Текущий каталог текущего диска — то место, где Windows-функции ищут файлы и подкаталоги, если полные пути в соответствующих параметрах не указаны. Например, если поток в процессе вызывает функцию *CreateFile*, чтобы открыть какой-нибудь файл, а полный путь не задан, система просматривает список файлов в текущем каталоге текущего диска. Этот каталог отслеживается самой системой, и, поскольку такая информация относится ко всему процессу, смена текущего диска или каталога одним из потоков распространяется и на остальные потоки в данном процессе.

Поток может получать и устанавливать текущие каталог и диск для процесса с помощью двух функций:

```
DWORD GetCurrentDirectory()  
    DWORD cchCurDir,  
    PTSTR pszCurDir);  
  
BOOL SetCurrentDirectory(PCTSTR pszCurDir);
```

Текущие каталоги для процесса

Система отслеживает текущие диск и каталог для процесса, но не текущие каталоги на каждом диске. Однако в операционной системе предусмотрен кое-какой сервис для манипуляций с текущими каталогами на разных дисках. Он реализуется через переменные окружения конкретного процесса. Например:

```
=C:=C:\Utility\Bin  
=D:=D:\Program Files
```

Эти переменные указывают, что текущим каталогом на диске С является \Utility\Bin, а на диске D — Program Files.

Если Вы вызываете функцию, передавая ей путь с именем диска, отличного от текущего, система сначала просматривает блок переменных окружения и пытается найти переменную, связанную с именем указанного диска. Если таковая есть, система выбирает текущий каталог на заданном диске в соответствии с ее значением, нет — текущим каталогом считается корневой.

Скажем, если текущий каталог для процесса — C:\Utility\Bin и Вы вызываете функцию *CreateFile*, чтобы открыть файл D:\ReadMe.txt, система ищет переменную **=D:**. Поскольку переменная **=D:** существует, система пытается открыть файл ReadMe.txt в каталоге D:\Program Files. А если бы таковой переменной не было, система искала бы файл ReadMe.txt в корневом каталоге диска D. Кстати, файловые Windows-функции никогда не добавляют и не изменяют переменные окружения, связанные с именами дисков, а лишь считывают их значения.



Для смены текущего каталога вместо Windows-функции *SetCurrentDirectory* можно использовать функцию *_chdir* из библиотеки С. Внутренне она тоже обращается к *SetCurrentDirectory*, но, кроме того, способна добавлять или модифицировать переменные окружения, что позволяет запоминать в программе текущие каталоги на различных дисках.

Если родительский процесс создает блок переменных окружения и хочет передать его дочернему процессу, тот не наследует текущие каталоги родительского процесса автоматически. Вместо этого у дочернего процесса текущими на всех дисках становятся корневые каталоги. Чтобы дочерний процесс унаследовал текущие каталоги родительского, последний должен создать соответствующие переменные окружения (и сделать это до порождения другого процесса). Родительский процесс может узнать, какие каталоги являются текущими, вызвав *GetFullPathName*:

```
DWORD GetFullPathName(
    PCTSTR pszFile,
    DWORD cchPath,
    PTSTR pszPath,
    PTSTR *ppszFilePart);
```

Например, чтобы получить текущий каталог на диске С, функцию вызывают так:

```
TCHAR szCurDir[MAX_PATH];
DWORD GetFullPathName(TEXT("C:\\"), MAX_PATH, szCurDir, NULL);
```

Не забывайте, что переменные окружения процесса должны всегда храниться в алфавитном порядке. Поэтому переменные, связанные с дисками, обычно приходится размещать в самом начале блока.

Определение версии системы

Весьма часто приложению требуется определять, в какой версии Windows оно выполняется. Причин тому несколько. Например, программа может использовать функции защиты, заложенные в Windows API, но в полной мере эти функции реализованы лишь в Windows 2000.

Насколько я помню, функция *GetVersion* есть в API всех версий Windows:

```
DWORD GetVersion();
```

С этой простой функцией связана целая история. Сначала ее разработали для 16-разрядной Windows, и она должна была в старшем слове возвращать номер версии MS-DOS, а в младшем — номер версии Windows. Соответственно в каждом слове старший байт сообщал основной номер версии, младший — дополнительный номер версии.

Увы, программист, писавший ее код, слегка ошибся, и получилось так, что номера версии Windows поменялись местами: в старший байт попадал дополнительный номер, а в младший — основной. Поскольку многие программисты уже начали пользоваться этой функцией, Microsoft пришлось оставить все, как есть, и изменить документацию с учетом ошибки.

Из-за всей этой неразберихи вокруг *GetVersion* в Windows API включили новую функцию — *GetVersionEx*:

```
BOOL GetVersionEx(POSVERSIONINFO pVersionInformation);
```

Перед обращением к *GetVersionEx* программа должна создать структуру OSVERSIONINFOEX, показанную ниже, и передать ее адрес этой функции.

```
typedef struct {
    DWORD dwOSVersionInfoSize;
    DWORD dwMajorVersion;
    DWORD dwMinorVersion;
    DWORD dwBuildNumber;
    DWORD dwPlatformId;
    TCHAR szCSDVersion[128];
    WORD wServicePackMajor;
    WORD wServicePackMinor;
    WORD wSuiteMask;
    BYTE wProductType;
    BYTE wReserved;
} OSVERSIONINFOEX, *POSVERSIONINFOEX;
```

Эта структура — новинка Windows 2000. В остальных версиях Windows используется структура OSVERSIONINFO, в которой нет последних пяти элементов, присутствующих в структуре OSVERSIONINFOEX.

Обратите внимание, что каждому компоненту номера версии операционной системы соответствует свой элемент структуры. Это сделано специально — чтобы программисты не возились с выборкой данных из всяких там старших-младших байтовых слов (и не путались в них!); теперь программе гораздо проще сравнивать ожидаемый номер версии операционной системы с действительным. Назначение каждого элемента структуры OSVERSIONINFOEX описано в таблице 4-2.

Элемент	Описание
<i>dwOSVersionInfoSize</i>	Размер структуры; перед обращением к функции <i>GetVersionEx</i> должен быть заполнен вызовом <i>sizeof(OSVERSIONINFO)</i> или <i>sizeof(OSVERSIONINFOEX)</i>
<i>dwMajorVersion</i>	Основной номер версии операционной системы
<i>dwMinorVersion</i>	Дополнительный номер версии операционной системы
<i>dwBuildNumber</i>	Версия сборки данной системы
<i>dwPlatformId</i>	Идентификатор платформы, поддерживаемой данной системой; его возможные значения: VER_PLATFORM_WIN32s (Win32s), VER_PLATFORM_WIN32_WINDOWS (Windows 95/98), VER_PLATFORM_WIN32_NT (Windows NT или Windows 2000), VER_PLATFORM_WIN32_CEHH (Windows CE)
<i>szCSDVersion</i>	Этот элемент содержит текст — дополнительную информацию об установленной операционной системе
<i>wServicePackMajor</i>	Основной номер версии последнего установленного пакета исправлений (service pack)
<i>wServicePackMinor</i>	Дополнительный номер версии последнего установленного пакета исправлений

Таблица 4-2. Элементы структуры OSVERSIONINFOEX

продолжение

Элемент	Описание
<i>wSuiteMask</i>	Сообщает, какие программные пакеты (suites) доступны в системе; его возможные значения: VER_SUITE_SMALLBUSINESS, VER_SUITE_ENTERPRISE, VER_SUITE_BACKOFFICE, VER_SUITE_COMMUNICATIONS, VER_SUITE_TERMINAL, VER_SUITE_SMALLBUSINESS_RESTRICTED, VER_SUITE_EMBEDDEDNT, VER_SUITE_DATACENTER
<i>wProductType</i>	Сообщает, какой именно вариант операционной системы установлен; его возможные значения: VER_NT_WORKSTATION, VER_NT_SERVER, VER_NT_DOMAIN_CONTROLLER
<i>wReserved</i>	Зарезервирован на будущее

В Windows 2000 появилась новая функция, *VerifyVersionInfo*, которая сравнивает версию установленной операционной системы с тем, что требует Ваше приложение:

```
BOOL VerifyVersionInfo(
    POSVERSIONINFOEX pVersionInformation,
    DWORD dwTypeMask,
    DWORDLONG dwlConditionMask);
```

Чтобы использовать эту функцию, создайте структуру OSVERSIONINFOEX, запишите в ее элемент *dwOSVersionInfoSize* размер структуры, а потом инициализируйте любые другие элементы, важные для Вашей программы. При вызове *VerifyVersionInfo* параметр *dwTypeMask* указывает, какие элементы структуры Вы инициализировали. Этот параметр принимает любые комбинации следующих флагов: VER_MINORVERSION, VER_MAJORVERSION, VER_BUILDNUMBER, VER_PLATFORMID, VER_SERVICEPACKMINOR, VER_SERVICEPACKMAJOR, VER_SUITENAME и VER_PRODUCT_TYPE. Последний параметр, *dwlConditionMask*, является 64-разрядным значением, которое управляет тем, как именно функция сравнивает информацию о версии системы с нужными Вам данными.

Параметр *dwlConditionMask* устанавливает правила сравнения через сложный набор битовых комбинаций. Для создания требуемой комбинации используйте макрос *VER_SET_CONDITION*:

```
VER_SET_CONDITION(
    DWORDLONG dwlConditionMask,
    ULONG dwTypeBitMask,
    ULONG dwConditionMask)
```

Первый параметр, *dwlConditionMask*, идентифицирует переменную, битами которой Вы манипулируете. Вы не передаете адрес этой переменной, потому что *VER_SET_CONDITION* — макрос, а не функция. Параметр *dwTypeBitMask* указывает один элемент в структуре OSVERSIONINFOEX, который Вы хотите сравнить со своими данными. (Для сравнения нескольких элементов придется обращаться к *VER_SET_CONDITION* несколько раз подряд.) Флаги, передаваемые в этом параметре, идентичны передаваемым в параметре *dwTypeMask* функции *VerifyVersionInfo*.

Последний параметр макроса `VER_SET_CONDITION`, *dwConditionMask*, сообщает, как Вы хотите проводить сравнение. Он принимает одно из следующих значений: `VER_EQUAL`, `VER_GREATER`, `VER_GREATER_EQUAL`, `VER_LESS` или `VER_LESS_EQUAL`. Вы можете использовать эти значения в сравнениях по `VER_PRODUCT_TYPE`. Например, значение `VER_NT_WORKSTATION` меньше, чем `VER_NT_SERVER`. Но в сравнениях по `VER_SUITENAME` вместо этих значений применяется `VER_AND` (должны быть установлены все программные пакеты) или `VER_OR` (должен быть установлен хотя бы один из программных пакетов).

Подготовив набор условий, Вы вызываете `VerifyVersionInfo` и получаете ненулевое значение, если система отвечает требованиям Вашего приложения, или 0, если она не удовлетворяет этим требованиям или если Вы неправильно вызвали функцию. Чтобы определить, почему `VerifyVersionInfo` вернула 0, вызовите `GetLastError`. Если та вернет `ERROR_OLD_WIN_VERSION`, значит, Вы правильно вызвали функцию `VerifyVersionInfo`, но система не соответствует предъявленным требованиям.

Вот как проверить, установлена ли Windows 2000:

```
// готовим структуру OSVERSIONINFOEX, сообщая, что нам нужна Windows 2000
OSVERSIONINFOEX osver = { 0 };
osver.dwOSVersionInfoSize = sizeof(osver);
osver.dwMajorVersion = 5;
osver.dwMinorVersion = 0;
osver.dwPlatformId = VER_PLATFORM_WIN32_NT;

// формируем маску условий
DWORDLONG dwlConditionMask = 0;      // всегда инициализируйте этот элемент так
VER_SET_CONDITION(dwlConditionMask, VER_MAJORVERSION, VER_EQUAL);
VER_SET_CONDITION(dwlConditionMask, VER_MINORVERSION, VER_EQUAL);
VER_SET_CONDITION(dwlConditionMask, VER_PLATFORMID, VER_EQUAL);

// проверяем версию
if (VerifyVersionInfo(&osver, VER_MAJORVERSION | VER_MINORVERSION | VER_PLATFORMID,
    dwlConditionMask)) {
    // хост-система точно соответствует Windows 2000
} else {
    // хост-система не является Windows 2000
}
```

Функция *CreateProcess*

Процесс создается при вызове Вашим приложением функции *CreateProcess*:

```
BOOL CreateProcess(
    PCTSTR pszApplicationName,
    PTSTR pszCommandLine,
    PSECURITY_ATTRIBUTES psaProcess,
    PSECURITY_ATTRIBUTES psaThread,
    BOOL bInheritHandles,
    DWORD fdwCreate,
    PVOID pvEnvironment,
    PCTSTR pszCurDir,
    PSTARTUPINFO psiStartInfo,
    PPROCESS_INFORMATION ppiProcInfo);
```

Когда поток в приложении вызывает *CreateProcess*, система создает объект ядра «процесс» с начальным значением счетчика числа его пользователей, равным 1. Этот объект — не сам процесс, а компактная структура данных, через которую операционная система управляет процессом. (Объект ядра «процесс» следует рассматривать как структуру данных со статистической информацией о процессе.) Затем система создает для нового процесса виртуальное адресное пространство и загружает в него код и данные как для исполняемого файла, так и для любых DLL (если таковые требуются).

Далее система формирует объект ядра «поток» (со счетчиком, равным 1) для первичного потока нового процесса. Как и в первом случае, объект ядра «поток» — это компактная структура данных, через которую система управляет потоком. Первичный поток начинает с исполнения стартового кода из библиотеки С/С++, который в конечном счете вызывает функцию *WinMain*, *wWinMain*, *main* или *wmain* в Вашей программе. Если системе удастся создать новый процесс и его первичный поток, *CreateProcess* вернет TRUE.



CreateProcess возвращает TRUE до окончательной инициализации процесса. Это означает, что на данном этапе загрузчик операционной системы еще не искал все необходимые DLL. Если он не сможет найти хотя бы одну из DLL или корректно провести инициализацию, процесс завершится. Но, поскольку *CreateProcess* уже вернула TRUE, родительский процесс ничего не узнает об этих проблемах.

На этом мы закончим общее описание и перейдем к подробному рассмотрению параметров функции *CreateProcess*.

Параметры *pszApplicationName* и *pszCommandLine*

Эти параметры определяют имя исполняемого файла, которым будет пользоваться новый процесс, и командную строку, передаваемую этому процессу. Начнем с *pszCommandLine*.



Обратите внимание на тип параметра *pszCommandLine*: PTSTR. Он означает, что *CreateProcess* ожидает передачи адреса строки, которая не является константой. Дело в том, что *CreateProcess* в процессе своего выполнения модифицирует переданную командную строку, но перед возвратом управления восстанавливает ее.

Это очень важно: если командная строка содержится в той части образа Вашего файла, которая предназначена только для чтения, возникнет ошибка доступа. Например, следующий код приведет к такой ошибке, потому что Visual C++ 6.0 поместит строку «NOTEPAD» в память только для чтения:

```
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
CreateProcess(NULL, TEXT("NOTEPAD"), NULL, NULL,
    FALSE, 0, NULL, NULL, &si, &pi);
```

Когда *CreateProcess* попытается модифицировать строку, произойдет ошибка доступа. (В прежних версиях Visual C++ эта строка была бы размещена в памяти для чтения и записи, и вызовы *CreateProcess* не приводили бы к ошибкам доступа.)

см. след. стр.

Лучший способ решения этой проблемы — перед вызовом *CreateProcess* копировать константную строку во временный буфер:

```
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
TCHAR szCommandLine[] = TEXT("NOTEPA");
CreateProcess(NULL, szCommandLine, NULL, NULL,
    FALSE, 0, NULL, &si, &pi);
```

Возможно, Вас заинтересуют ключи /Gf и /GF компилятора Visual C++, которые исключают дублирование строк и запрещают их размещение в области только для чтения. (Также обратите внимание на ключ /ZI, который позволяет задействовать отладочную функцию Edit & Continue, поддерживаемую Visual Studio, и подразумевает активизацию ключа /GF.) В общем, лучшее, что можете сделать Вы, — использовать ключ /GF или создать временный буфер. А еще лучше, если Microsoft исправит функцию *CreateProcess*, чтобы та не морочила нам голову. Надеюсь, в следующей версии Windows так и будет.

Кстати, при вызове ANSI-версии *CreateProcess* в Windows 2000 таких проблем нет, поскольку в этой версии функции командная строка копируется во временный буфер (см. главу 2).

Параметр *pszCommandLine* позволяет указать полную командную строку, используемую функцией *CreateProcess* при создании нового процесса. Разбирая эту строку, функция полагает, что первый компонент в ней представляет собой имя исполняемого файла, который Вы хотите запустить. Если в имени этого файла не указано расширение, она считает его EXE. Далее функция приступает к поиску заданного файла и делает это в следующем порядке:

1. Каталог, содержащий EXE-файл вызывающего процесса.
2. Текущий каталог вызывающего процесса.
3. Системный каталог Windows.
4. Основной каталог Windows.
5. Каталоги, перечисленные в переменной окружения PATH.

Конечно, если в имени файла указан полный путь доступа, система сразу обращается туда и не просматривает эти каталоги. Найдя нужный исполняемый файл, она создает новый процесс и проецирует код и данные исполняемого файла на адресное пространство этого процесса. Затем обращается к процедурам стартового кода из библиотеки C/C++. Тот в свою очередь, как уже говорилось, анализирует командную строку процесса и передает (*w*)*WinMain* адрес первого (за именем исполняемого файла) аргумента как *pszCmdLine*.

Все, о чем я сказал, произойдет, только если параметр *pszApplicationName* равен NULL (что и бывает в 99% случаев). Вместо NULL можно передать адрес строки с именем исполняемого файла, который надо запустить. Однако тогда придется указать не только его имя, но и расширение, поскольку в этом случае имя не дополняется расширением EXE автоматически. *CreateProcess* предполагает, что файл находится в текущем каталоге (если полный путь не задан). Если в текущем каталоге файла нет, функция не станет искать его в других каталогах, и на этом все закончится.

Но даже при указанном в *pszApplicationName* имени файла *CreateProcess* все равно передает новому процессу содержимое параметра *pszCommandLine* как командную строку. Допустим, Вы вызвали *CreateProcess* так:

```
// размещаем строку пути в области памяти для чтения и записи
TCHAR szPath[ ] = TEXT("WORDPAD README.TXT");

// порождаем новый процесс
CreateProcess(TEXT("C:\\WINNT\\SYSTEM32\\NOTEPAD.EXE"), szPath, ...);
```

Система запускает Notepad, а в его командной строке мы видим «WORDPAD README.TXT». Странно, да? Но так уж она работает, эта функция *CreateProcess*. Упомянутая возможность, которую обеспечивает параметр *pszApplicationName*, на самом деле введена в *CreateProcess* для поддержки подсистемы POSIX в Windows 2000.

Параметры *psaProcess*, *psaThread* и *bInheritHandles*

Чтобы создать новый процесс, система должна сначала создать объекты ядра «процесс» и «поток» (для первичного потока процесса). Поскольку это объекты ядра, родительский процесс получает возможность связать с ними атрибуты защиты. Параметры *psaProcess* и *psaThread* позволяют определить нужные атрибуты защиты для объектов «процесс» и «поток» соответственно. В эти параметры можно занести NULL, и система закрепит за данными объектами дескрипторы защиты по умолчанию. В качестве альтернативы можно объявить и инициализировать две структуры SECURITY_ATTRIBUTES; тем самым Вы создадите и присвоите объектам «процесс» и «поток» свои атрибуты защиты.

Структуры SECURITY_ATTRIBUTES для параметров *psaProcess* и *psaThread* используются и для того, чтобы какой-либо из этих двух объектов получил статус наследуемого любым дочерним процессом. (О теории, на которой построено наследование описателей объектов ядра, я рассказывал в главе 3.)

Короткая программа на рис. 4-2 демонстрирует, как наследуются описатели объектов ядра. Будем считать, что процесс А порождает процесс В и заносит в параметр *psaProcess* адрес структуры SECURITY_ATTRIBUTES, в которой элемент *bInheritHandle* установлен как TRUE. Одновременно параметр *psaThread* указывает на другую структуру SECURITY_ATTRIBUTES, в которой значение элемента *bInheritHandle* — FALSE.

Создавая процесс В, система формирует объекты ядра «процесс» и «поток», а затем — в структуре, на которую указывает параметр *ppriProcInfo* (о нем поговорим позже), — возвращает их описатели процессу А, и с этого момента тот может манипулировать только что созданными объектами «процесс» и «поток».

Теперь предположим, что процесс А собирается вторично вызвать функцию *CreateProcess*, чтобы породить процесс С. Сначала ему нужно определить, стоит ли предоставить процессу С доступ к своим объектам ядра. Для этого используется параметр *bInheritHandles*. Если он приравнен TRUE, система передает процессу С все наследуемые описатели. В этом случае наследуется и описатель объекта ядра «процесс» процесса В. А вот описатель объекта «первичный поток» процесса В не наследуется ни при каком значении *bInheritHandles*. Кроме того, если процесс А вызывает *CreateProcess*, передавая через параметр *bInheritHandles* значение FALSE, процесс С не наследует никаких описателей, используемых в данный момент процессом А.

Inherit.c

```
*****  
Модуль: Inherit.c  
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)  
*****  
  
#include <Windows.h>  
  
int WINAPI WinMain(HINSTANCE hinstExe, HINSTANCE,  
PSTR pszCmdLine, int nCmdShow) {  
  
    // готовим структуру STARTUPINFO для создания процессов  
    STARTUPINFO si = { sizeof(si) };  
    SECURITY_ATTRIBUTES saProcess, saThread;  
    PROCESS_INFORMATION piProcessB, piProcessC;  
    TCHAR szPath[MAX_PATH];  
  
    // готовимся к созданию процесса В из процесса А;  
    // описатель, идентифицирующий новый объект "процесс",  
    // должен быть наследуемым  
    saProcess.nLength = sizeof(saProcess);  
    saProcess.lpSecurityDescriptor = NULL;  
    saProcess.bInheritHandle = TRUE;  
  
    // описатель, идентифицирующий новый объект "поток",  
    // НЕ должен быть наследуемым  
    saThread.nLength = sizeof(saThread);  
    saThread.lpSecurityDescriptor = NULL;  
    saThread.bInheritHandle = FALSE;  
  
    // порождаем процесс В  
    lstrcpy(szPath, TEXT("ProcessB"));  
    CreateProcess(NULL, szPath, &saProcess, &saThread,  
        FALSE, 0, NULL, NULL, &si, &piProcessB);  
  
    // структура pi содержит два описателя, относящиеся к процессу А:  
    // hProcess, который идентифицирует объект "процесс" процесса В  
    // и является наследуемым, и hThread, который идентифицирует объект  
    // "первичный поток" процесса В и НЕ является наследуемым  
  
    // готовимся создать процесс С из процесса А;  
    // так как в psaProcess и psaThread передаются NULL, описатели  
    // объектов "процесс" и "первичный поток" процесса С считаются  
    // ненаследуемыми по умолчанию  
  
    // если процесс А создаст еще один процесс, тот НЕ унаследует  
    // описатели объектов "процесс" и "первичный поток" процесса С  
  
    // поскольку в параметре bInheritHandles передается TRUE,  
    // процесс С унаследует описатель, идентифицирующий объект  
    // "процесс" процесса В, но НЕ описатель, идентифицирующий объект
```

Рис. 4-2. Пример, иллюстрирующий наследование описателей объектов ядра

Рис. 4-2. продолжение

```

// "первичный поток" того же процесса
lstrcpy(szPath, TEXT("ProcessC"));
CreateProcess(NULL, szPath, NULL, NULL,
    TRUE, 0, NULL, NULL, &si, &piProcessC);

return(0);
}

```

Параметр *fdwCreate*

Параметр *fdwCreate* определяет флаги, влияющие на то, как именно создается новый процесс. Флаги комбинируются булевым оператором OR.

- Флаг DEBUG_PROCESS дает возможность родительскому процессу проводить отладку дочернего, а также всех процессов, которые последним могут быть порождены. Если этот флаг установлен, система уведомляет родительский процесс (он теперь получает статус отладчика) о возникновении определенных событий в любом из дочерних процессов (а они получают статус отлаживаемых).
- Флаг DEBUG_ONLY_THIS_PROCESS аналогичен флагу DEBUG_PROCESS с тем исключением, что заставляет систему уведомлять родительский процесс о возникновении специфических событий только в одном дочернем процессе — его прямом потомке. Тогда, если дочерний процесс создаст ряд дополнительных, отладчик уже не уведомляется о событиях, «происходящих» в них.
- Флаг CREATE_SUSPENDED позволяет создать процесс и в то же время приостановить его первичный поток. Это позволяет родительскому процессу модифицировать содержимое памяти в адресном пространстве дочернего, изменять приоритет его первичного потока или включать этот процесс в задание (job) до того, как он получит шанс на выполнение. Внеся нужные изменения в дочерний процесс, родительский разрешает выполнение его кода вызовом функции *ResumeThread* (см. главу 7).
- Флаг DETACHED_PROCESS блокирует доступ процессу, иницииированному консольной программой, к созданному родительским процессом консольному окну и сообщает системе, что вывод следует перенаправить в новое окно. CUI-процесс, создаваемый другим CUI-процессом, по умолчанию использует консольное окно родительского процесса. (Вы, очевидно, заметили, что при запуске компилятора С из командного процессора новое консольное окно не создается; весь его вывод «подписывается» в нижнюю часть существующего консольного окна.) Таким образом, этот флаг заставляет новый процесс перенаправлять свой вывод в новое консольное окно.
- Флаг CREATE_NEW_CONSOLE приводит к созданию нового консольного окна для нового процесса. Имейте в виду, что одновременная установка флагов CREATE_NEW_CONSOLE и DETACHED_PROCESS недопустима.
- Флаг CREATE_NO_WINDOW не дает создавать никаких консольных окон для данного приложения и тем самым позволяет исполнять его без пользовательского интерфейса.

- Флаг CREATE_NEW_PROCESS_GROUP служит для модификации списка процессов, уведомляемых о нажатии клавиш Ctrl+C и Ctrl+Break. Если в системе одновременно исполняется несколько CUI-процессов, то при нажатии одной из упомянутых комбинаций клавиш система уведомляет об этом только процессы, включенные в группу. Указав этот флаг при создании нового CUI-процесса, Вы создаете и новую группу.
- Флаг CREATE_DEFAULT_ERROR_MODE сообщает системе, что новый процесс не должен наследовать режимы обработки ошибок, установленные в родительском (см. раздел, где я рассказывал о функции *SetErrorMode*).
- Флаг CREATE_SEPARATE_WOW_VDM полезен только при запуске 16-разрядного Windows-приложения в Windows 2000. Если он установлен, система создает отдельную виртуальную DOS-машину (Virtual DOS-machine, VDM) и запускает 16-разрядное Windows-приложение именно в ней. (По умолчанию все 16-разрядные Windows-приложения выполняются в одной, общей VDM.) Выполнение приложения в отдельной VDM дает большое преимущество: «рухнув», приложение уничтожит лишь эту VDM, а программы, выполняемые в других VDM, продолжат нормальную работу. Кроме того, 16-разрядные Windows-приложения, выполняемые в раздельных VDM, имеют и раздельные очереди ввода. Это значит, что, если одно приложение вдруг «зависнет», приложения в других VDM продолжат прием ввода. Единственный недостаток работы с несколькими VDM в том, что каждая из них требует значительных объемов физической памяти. Windows 98 выполняет все 16-разрядные Windows-приложения только в одной VDM, и изменить тут ничего нельзя.
- Флаг CREATE_SHARED_WOW_VDM полезен только при запуске 16-разрядного Windows-приложения в Windows 2000. По умолчанию все 16-разрядные Windows-приложения выполняются в одной VDM, если только не указан флаг CREATE_SEPARATE_WOW_VDM. Однако стандартное поведение Windows 2000 можно изменить, присвоив значение «yes» параметру DefaultSeparateVDM в разделе HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\WOW. (После модификации этого параметра систему надо перезагрузить.) Установив значение «yes», но указав флаг CREATE_SHARED_WOW_VDM, Вы вновь заставите Windows 2000 выполнять все 16-разрядные Windows-приложения в одной VDM.
- Флаг CREATE_UNICODE_ENVIRONMENT сообщает системе, что блок переменных окружения дочернего процесса должен содержать Unicode-символы. По умолчанию блок формируется на основе ANSI-символов.
- Флаг CREATE_FORCEDOS заставляет систему выполнять программу MS-DOS,строенную в 16-разрядное приложение OS/2.
- Флаг CREATE_BREAKAWAY_FROM_JOB позволяет процессу, включенному в задание, создать новый процесс, отделенный от этого задания (см. главу 5).

Параметр *fdwCreate* разрешает задать и класс приоритета процесса. Однако это необязательно и даже, как правило, не рекомендуется; система присваивает новому процессу класс приоритета по умолчанию. Возможные классы приоритета перечислены в следующей таблице.

Класс приоритета	Флаговый идентификатор
Idle (простаивающий)	IDLE_PRIORITY_CLASS
Below normal (ниже обычного)	BELOW_NORMAL_PRIORITY_CLASS
Normal (обычный)	NORMAL_PRIORITY_CLASS
Above normal (выше обычного)	ABOVE_NORMAL_PRIORITY_CLASS
High (высокий)	HIGH_PRIORITY_CLASS
Realtime (реального времени)	REALTIME_PRIORITY_CLASS

Классы приоритета влияют на распределение процессорного времени между процессами и их потоками. (Подробнее на эту тему см. главу 7.)



Классы приоритета BELOW_NORMAL_PRIORITY_CLASS и ABOVE_NORMAL_PRIORITY_CLASS введены лишь в Windows 2000; они не поддерживаются в Windows NT 4.0, Windows 95 или Windows 98.

Параметр *pvEnvironment*

Параметр *pvEnvironment* указывает на блок памяти, хранящий строки переменных окружения, которыми будет пользоваться новый процесс. Обычно вместо этого параметра передается NULL, в результате чего дочерний процесс наследует строки переменных окружения от родительского процесса. В качестве альтернативы можно вызвать функцию *GetEnvironmentStrings*:

```
PVOID GetEnvironmentStrings();
```

Она позволяет узнать адрес блока памяти со строками переменных окружения, используемых вызывающим процессом. Полученный адрес можно занести в параметр *pvEnvironment* функции *CreateProcess*. (Именно это и делает *CreateProcess*, если Вы передаете ей NULL вместо *pvEnvironment*.) Если этот блок памяти Вам больше не нужен, освободите его, вызвав функцию *FreeEnvironmentStrings*:

```
BOOL FreeEnvironmentStrings(PTSTR pszEnvironmentBlock);
```

Параметр *pszCurDir*

Он позволяет родительскому процессу установить текущие диск и каталог для дочернего процесса. Если его значение — NULL, рабочий каталог нового процесса будет тем же, что и у приложения, его породившего. А если он отличен от NULL, то должен указывать на строку (с нулевым символом в конце), содержащую нужный диск и каталог. Заметьте, что в путь надо включать и букву диска.

Параметр *psiStartInfo*

Этот параметр указывает на структуру STARTUPINFO:

```
typedef struct _STARTUPINFO {
    DWORD cb;
    PSTR lpReserved;
    PSTR lpDesktop;
    PSTR lpTitle;
    DWORD dwX;
    DWORD dwY;
```

см. след. стр.

```
DWORD dwXSize;
DWORD dwYSize;
DWORD dwXCountChars;
DWORD dwYCountChars;
DWORD dwFillAttribute;
DWORD dwFlags;
WORD wShowWindow;
WORD cbReserved2;
PBYTE lpReserved2;
HANDLE hStdInput;
HANDLE hStdOutput;
HANDLE hStdError;
} STARTUPINFO, *LPSTARTUPINFO;
```

Элементы структуры `STARTUPINFO` используются Windows-функциями при создании нового процесса. Надо сказать, что большинство приложений порождает процессы с атрибутами по умолчанию. Но и в этом случае Вы должны инициализировать все элементы структуры `STARTUPINFO` хотя бы нулевыми значениями, а в элемент `cb` — заносить размер этой структуры:

```
STARTUPINFO si = { sizeof(si) };
CreateProcess(..., &si, ...);
```

К сожалению, разработчики приложений часто забывают о необходимости инициализации этой структуры. Если Вы не обнулите ее элементы, в них будет содержаться мусор, оставшийся в стеке вызывающего потока. Функция `CreateProcess`, получив такую структуру данных, либо создаст новый процесс, либо нет — все зависит от того, что именно окажется в этом мусоре.

Когда Вам понадобится изменить какие-то элементы структуры, делайте это перед вызовом `CreateProcess`. Все элементы этой структуры подробно рассматриваются в таблице 4-3. Но заметьте, что некоторые элементы имеют смысл, только если до-чернее приложение создает перекрываемое (overlapped) окно, а другие — если это приложение осуществляет ввод-вывод на консоль.

Элемент	Окно или консоль	Описание
<i>cb</i>	То и другое	Содержит количество байтов, занимаемых структурой <code>STARTUPINFO</code> . Служит для контроля версий — на тот случай, если Microsoft расширит эту структуру в будущем. Программа должна инициализировать <i>cb</i> как <code>sizeof(STARTUPINFO)</code> .
<i>lpReserved</i>	То и другое	Зарезервирован. Инициализируйте как NULL.
<i>lpDesktop</i>	То и другое	Идентифицирует имя рабочего стола, на котором запускается приложение. Если указанный рабочий стол существует, новый процесс сразу же связывается с ним. В ином случае система сначала создает рабочий стол с атрибутами по умолчанию, присваивает ему имя, указанное в данном элементе структуры, и связывает его с новым процессом. Если <i>lpDesktop</i> равен NULL (что чаще всего и бывает), процесс связывается с текущим рабочим столом.
<i>lpTitle</i>	Консоль	Определяет заголовок консольного окна. Если <i>lpTitle</i> — NULL, в заголовок выводится имя исполняемого файла.

Таблица 4-3. Элементы структуры `STARTUPINFO`

продолжение

Элемент	Окно или консоль	Описание
<i>dwX</i> <i>dwY</i>	То и другое	Указывают <i>x</i> - и <i>y</i> -координаты (в пикселях) окна приложения. Эти координаты используются, только если дочерний процесс создает свое первое перекрываемое окно с идентификатором CW_USEDEFAULT в параметре <i>x</i> функции <i>CreateWindow</i> . В приложениях, создающих консольные окна, данные элементы определяют верхний левый угол консольного окна.
<i>dwXSize</i> <i>dwYSize</i>	То и другое	Определяют ширину и высоту (в пикселях) окна приложения. Эти значения используются, только если дочерний процесс создает свое первое перекрываемое окно с идентификатором CW_USEDEFAULT в параметре <i>nWidth</i> функции <i>CreateWindow</i> . В приложениях, создающих консольные окна, данные элементы определяют ширину и высоту консольного окна.
<i>dwXCountChars</i> <i>dwYCountChars</i>	Консоль	Определяют ширину и высоту (в символах) консольных окон дочернего процесса.
<i>dwFillAttribute</i>	Консоль	Задает цвет текста и фона в консольных окнах дочернего процесса.
<i>dwFlags</i>	То и другое	См. ниже и следующую таблицу.
<i>wShowWindow</i>	Окно	Определяет, как именно должно выглядеть первое перекрываемое окно дочернего процесса, если приложение при первом вызове функции <i>ShowWindow</i> передает в параметре <i>nCmdShow</i> идентификатор SW_SHOWDEFAULT. В этот элемент можно записать любой из идентификаторов типа SW_*, обычно используемых при вызове <i>ShowWindow</i> .
<i>cbReserved2</i>	То и другое	Зарезервирован. Инициализируйте как 0.
<i>lpReserved2</i>	То и другое	Зарезервирован. Инициализируйте как NULL.
<i>bStdInput</i> <i>bStdOutput</i> <i>bStdError</i>	Консоль	Определяют описатели буферов для консольного ввода-вывода. По умолчанию <i>bStdInput</i> идентифицирует буфер клавиатуры, а <i>bStdOutput</i> и <i>bStdError</i> — буфер консольного окна.

Теперь, как я и обещал, обсудим элемент *dwFlags*. Он содержит набор флагов, позволяющих управлять созданием дочернего процесса. Большая часть флагов просто сообщает функции *CreateProcess*, содержит ли прочие элементы структуры STARTUPINFO полезную информацию или некоторые из них можно игнорировать. Список допустимых флагов приведен в следующей таблице.

Флаг	Описание
STARTF_USESIZE	Заставляет использовать элементы <i>dwXSize</i> и <i>dwYSize</i>
STARTF_USESHOWWINDOW	Заставляет использовать элемент <i>wShowWindow</i>
STARTF_USEPOSITION	Заставляет использовать элементы <i>dwX</i> и <i>dwY</i>
STARTF_USECOUNTCHARS	Заставляет использовать элементы <i>dwXCountChars</i> и <i>dwYCountChars</i>
STARTF_USEFILLATTRIBUTE	Заставляет использовать элемент <i>dwFillAttribute</i>
STARTF_USESTDHANDLES	Заставляет использовать элементы <i>bStdInput</i> , <i>bStdOutput</i> и <i>bStdError</i>

см. след. стр.

продолжение

Флаг	Описание
STARTF_RUN_FULLSCREEN	Приводит к тому, что консольное приложение на компьютере с процессором типа <i>x86</i> запускается в полноэкранном режиме

Два дополнительных флага — STARTF_FORCEONFEEDBACK и STARTF_FORCEOFFFEEDBACK — позволяют контролировать форму курсора мыши в момент запуска нового процесса. Поскольку Windows поддерживает истинную вытесняющую многозадачность, можно запустить одно приложение и, пока оно инициализируется, поработать с другой программой. Для визуальной обратной связи с пользователем функция *CreateProcess* временно изменяет форму системного курсора мыши:



Курсор такой формы подсказывает: можно либо подождать чего-нибудь, что вот-вот случится, либо продолжить работу в системе. Если же Вы укажете флаг STARTF_FORCEOFFFEEDBACK, *CreateProcess* не станет добавлять «песочные часы» к стандартной стрелке.

Флаг STARTF_FORCEONFEEDBACK заставляет *CreateProcess* отслеживать инициализацию нового процесса и в зависимости от результата проверки изменять форму курсора. Когда функция *CreateProcess* вызывается с этим флагом, курсор преобразуется в «песочные часы». Если спустя две секунды от нового процесса не поступает GUI-вызов, она восстанавливает исходную форму курсора.

Если же в течение двух секунд процесс все же делает GUI-вызов, *CreateProcess* ждет, когда приложение откроет свое окно. Это должно произойти в течение пяти секунд после GUI-вызова. Если окно не появилось, *CreateProcess* восстанавливает курсор, а появилось — сохраняет его в виде «песочных часов» еще на пять секунд. Как только приложение вызовет функцию *GetMessage*, сообщая тем самым, что оно закончило инициализацию, *CreateProcess* немедленно сменит курсор на стандартный и прекратит мониторинг нового процесса.

В заключение раздела — несколько слов об элементе *wShowWindow* структуры STARTUPINFO. Этот элемент инициализируется значением, которое Вы передаете в *(w)WinMain* через ее последний параметр, *nCmdShow*. Он позволяет указать, в каком виде должно появиться главное окно Вашего приложения. В качестве значения используется один из идентификаторов, обычно передаваемых в *ShowWindow* (чаще всего SW_SHOWNORMAL или SW_SHOWMINNOACTIVE, но иногда и SW_SHOWDEFAULT).

После запуска программы из Explorer ее функция *(w)WinMain* вызывается с SW_SHOWNORMAL в параметре *nCmdShow*. Если же Вы создаете для нее ярлык, то можете указать в его свойствах, в каком виде должно появляться ее главное окно. На рис. 4-3 показано окно свойств для ярлыка Notepad. Обратите внимание на список Run, в котором выбирается начальное состояние окна Notepad.

Когда Вы активизируете этот ярлык из Explorer, последний создает и инициализирует структуру STARTUPINFO, а затем вызывает *CreateProcess*. Это приводит к запуску Notepad, а его функция *(w)WinMain* получает SW_SHOWMINNOACTIVE в параметре *nCmdShow*.

Таким образом, пользователь может легко выбирать, в каком окне запускать программу — нормальном, свернутом или развернутом.

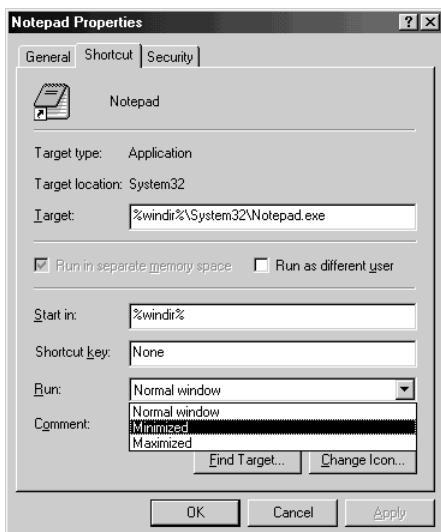


Рис. 4-3. Окно свойств для ярлыка Notepad

Наконец, чтобы получить копию структуры STARTUPINFO, инициализированной родительским процессом, приложение может вызвать:

```
VOID GetStartupInfo(PSTARTUPINFO pStartupInfo);
```

Анализируя эту структуру, дочерний процесс может изменять свое поведение в зависимости от значений ее элементов.



Хотя в документации на Windows об этом четко не сказано, перед вызовом *GetStartupInfo* нужно инициализировать элемент *cb* структуры STARTUPINFO:

```
STARTUPINFO si = { sizeof(si) };
GetStartupInfo(&si);
:
```

Параметр *ppiProcInfo*

Параметр *ppiProcInfo* указывает на структуру PROCESS_INFORMATION, которую Вы должны предварительно создать; ее элементы инициализируются самой функцией *CreateProcess*. Структура представляет собой следующее:

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
} PROCESS_INFORMATION;
```

Как я уже говорил, создание нового процесса влечет за собой создание объектов ядра «процесс» и «поток». В момент создания система присваивает счетчику каждого объекта начальное значение — единицу. Далее функция *CreateProcess* (перед самым возвратом управления) открывает объекты «процесс» и «поток» и заносит их описатели, специфичные для данного процесса, в элементы *hProcess* и *hThread* структуры PROCESS_INFORMATION. Когда *CreateProcess* открывает эти объекты, счетчики каждого из них увеличиваются до 2.

Это означает, что, перед тем как система сможет высвободить из памяти объект «процесс», процесс должен быть завершен (счетчик уменьшен до 1), а родительский процесс обязан вызвать функцию *CloseHandle* (и тем самым обнулить счетчик). То же самое относится и к объекту «поток»: поток должен быть завершен, а родительский процесс должен закрыть описатель объекта «поток». Подробнее об освобождении объектов «поток» см. раздел «Дочерние процессы» в этой главе.



Не забывайте закрывать описатели дочернего процесса и его первичного потока, иначе, пока Вы не закроете свое приложение, будет происходить утечка ресурсов. Конечно, система высвободит все эти ресурсы после завершения Вашего процесса, но хорошо написанная программа должна сама закрывать описатели дочернего процесса и его первичного потока, как только необходимость в них отпадает. Пропуск этой операции — одна из самых частых ошибок.

Почему-то многие разработчики считают, будто закрытие описателя процесса или потока заставляет систему уничтожить этот процесс или поток. Это абсолютно неправильно. Закрывая описатель, Вы просто сообщаете системе, что статистические данные для этого процесса или потока Вас больше не интересуют, но процесс или поток продолжает исполняться системой до тех пор, пока он сам не завершит себя.

Созданному объекту ядра «процесс» присваивается уникальный идентификатор; ни у каких других объектов этого типа в системе не может быть одинаковых идентификаторов. Это же касается и объектов ядра «поток». Причем идентификаторы процесса и потока тоже разные, и их значения никогда не бывают нулевыми. Завершая свою работу, *CreateProcess* заносит значения идентификаторов в элементы *dwProcessId* и *dwThreadId* структуры *PROCESS_INFORMATION*. Эти идентификаторы просто облегчают определение процессов и потоков в системе; их используют, как правило, лишь специализированные утилиты вроде Task Manager.

Подчеркну еще один чрезвычайно важный момент: система способна повторно использовать идентификаторы процессов и потоков. Например, при создании процесса система формирует объект «процесс», присваивая ему идентификатор со значением, допустим, 122. Создавая новый объект «процесс», система уже не присвоит ему данный идентификатор. Но после выгрузки из памяти первого объекта следующему создаваемому объекту «процесс» может быть присвоен тот же идентификатор — 122.

Эту особенность нужно учитывать при написании кода, избегая ссылок на неверный объект «процесс» (или «поток»). Действительно, затребовать и сохранить идентификатор процесса несложно, но задумайтесь, что получится, если в следующий момент этот процесс будет завершен, а новый получит тот же идентификатор: сохраненный ранее идентификатор уже связан совсем с другим процессом.

Иногда программе приходится определять свой родительский процесс. Однако родственные связи между процессами существуют лишь на стадии создания дочернего процесса. Непосредственно перед началом исполнения кода в дочернем процессе Windows перестает учитывать его родственные связи. В предыдущих версиях Windows не было функций, которые позволяли бы программе обращаться с запросом к ее родительскому процессу. Но ToolHelp-функции, появившиеся в современных версиях Windows, сделали это возможным. С этой целью Вы должны использовать структуру *PROCESSENTRY32*: ее элемент *th32ParentProcessID* возвращает идентификатор «родителя» данного процесса. Тем не менее, если Вашей программе нужно взаимодей-

ствовать с родительским процессом, от идентификаторов лучше отказаться. Почему — я уже говорил. Для определения родительского процесса существуют более надежные механизмы: объекты ядра, описатели окон и т. д.

Единственный способ добиться того, чтобы идентификатор процесса или потока не использовался повторно, — не допускать разрушения объекта ядра «процесс» или «поток». Если Вы только что создали новый процесс или поток, то можете просто не закрывать описатели на эти объекты — вот и все. А по окончании операций с идентификатором, вызовите функцию *CloseHandle* и освободите соответствующие объекты ядра. Однако для дочернего процесса этот способ не годится, если только он не унаследовал описатели объектов ядра от родительского процесса.

Завершение процесса

Процесс можно завершить четырьмя способами:

- входная функция первичного потока возвращает управление (рекомендуемый способ);
- один из потоков процесса вызывает функцию *ExitProcess* (нежелательный способ);
- поток другого процесса вызывает функцию *TerminateProcess* (тоже нежелательно);
- все потоки процесса умирают по своей воле (большая редкость).

В этом разделе мы обсудим только что перечисленные способы завершения процесса, а также рассмотрим, что на самом деле происходит в момент его окончания.

Возврат управления входной функцией первичного потока

Приложение следует проектировать так, чтобы его процесс завершался только после возврата управления входной функцией первичного потока. Это единственный способ, гарантирующий корректную очистку всех ресурсов, принадлежавших первичному потоку. При этом:

- любые C++-объекты, созданные данным потоком, уничтожаются соответствующими деструкторами;
- система освобождает память, которую занимал стек потока;
- система устанавливает код завершения процесса (поддерживаемый объектом ядра «процесс») — его и возвращает Ваша входная функция;
- счетчик пользователей данного объекта ядра «процесс» уменьшается на 1.

Функция *ExitProcess*

Процесс завершается, когда один из его потоков вызывает *ExitProcess*:

```
VOID ExitProcess(UINT fuExitCode);
```

Эта функция завершает процесс и заносит в параметр *fuExitCode* код завершения процесса. Возвращаемого значения у *ExitProcess* нет, так как результат ее действия — завершение процесса. Если за вызовом этой функции в программе присутствует какой-нибудь код, он никогда не исполняется.

Когда входная функция (*WinMain*, *wWinMain*, *main* или *wmain*) в Вашей программе возвращает управление, оно передается стартовому коду из библиотеки C/C++, и тот проводит очистку всех ресурсов, выделенных им процессу, а затем обращается к

ExitProcess, передавая ей значение, возвращенное входной функцией. Вот почему возврат управления входной функцией первичного потока приводит к завершению всего процесса. Обратите внимание, что при завершении процесса прекращается выполнение и всех других его потоков.

Кстати, в документации из Platform SDK утверждается, что процесс не завершается до тех пор, пока не завершится выполнение всех его потоков. Это, конечно, верно, но тут есть одна тонкость. Стартовый код из библиотеки C/C++ обеспечивает завершение процесса, вызывая *ExitProcess* после того, как первичный поток Вашего приложения возвращается из входной функции. Однако, вызвав из нее функцию *ExitThread* (вместо того чтобы вызвать *ExitProcess* или просто вернуть управление), Вы завершили первый поток, но не сам процесс — если в нем еще выполняется какой-то другой поток (или потоки).

Заметьте, что такой вызов *ExitProcess* или *ExitThread* приводит к уничтожению процесса или потока, когда выполнение функции еще не завершилось. Что касается операционной системы, то здесь все в порядке: она корректно очистит все ресурсы, выделенные процессу или потоку. Но в приложении, написанном на C/C++, следует избегать вызова этих функций, так как библиотеке C/C++ скорее всего не удастся провести должную очистку. Взгляните на этот код:

```
#include <windows.h>
#include <stdio.h>

class CSomeObj {
public:
    CSomeObj() { printf("Constructor\r\n"); }
    ~CSomeObj() { printf("Destructor\r\n"); }
};

CSomeObj g_GlobalObj;

void main () {
    CSomeObj LocalObj;
    ExitProcess(0);           // этого здесь не должно быть

    // в конце этой функции компилятор автоматически вставил код
    // для вызова деструктора LocalObj, но ExitProcess не дает его выполнить
}
```

При его выполнении Вы увидите:

```
Constructor
Constructor
```

Код конструирует два объекта: глобальный и локальный. Но Вы никогда не увидите строку *Destructor*. C++-объекты не разрушаются должным образом из-за того, что *ExitProcess* форсирует уничтожение процесса и библиотека C/C++ не получает шанса на очистку.

Как я уже говорил, никогда не вызывайте *ExitProcess* в явном виде. Если я уберу из предыдущего примера вызов *ExitProcess*, программа выведет такие строки:

```
Constructor
Constructor
Destructor
Destructor
```

Простой возврат управления от входной функции первичного потока позволил библиотеке C/C++ провести нужную очистку и корректно разрушить C++-объекты. Кстати, все, о чем я рассказал, относится не только к объектам, но и ко многим другим вещам, которые библиотека C/C++ делает для Вашего процесса.



Явные вызовы ExitProcess и ExitThread — распространенная ошибка, которая мешает правильной очистке ресурсов. В случае ExitThread процесс продолжает работать, но при этом весьма вероятна утечка памяти или других ресурсов.

Функция *TerminateProcess*

Вызов функции *TerminateProcess* тоже завершает процесс:

```
BOOL TerminateProcess(
    HANDLE hProcess,
    UINT fuExitCode);
```

Главное отличие этой функции от *ExitProcess* в том, что ее может вызвать любой поток и завершить любой процесс. Параметр *hProcess* идентифицирует описатель завершаемого процесса, а в параметре *fuExitCode* возвращается код завершения процесса.

Пользуйтесь *TerminateProcess* лишь в том случае, когда иным способом завершить процесс не удается. Процесс не получает абсолютно никаких уведомлений о том, что он завершается, и приложение не может ни выполнить очистку, ни предотвратить свое неожиданное завершение (если оно, конечно, не использует механизмы защиты). При этом теряются все данные, которые процесс не успел переписать из памяти на диск.

Процесс действительно не имеет ни малейшего шанса самому провести очистку, но операционная система высвобождает все принадлежавшие ему ресурсы: возвращает себе выделенную им память, закрывает любые открытые файлы, уменьшает счетчики соответствующих объектов ядра и разрушает все его User- и GDI-объекты.

По завершении процесса (не важно каким способом) система гарантирует: после него ничего не останется — даже намеков на то, что он когда-то выполнялся. *Завершенный процесс не оставляет за собой никаких следов*. Надеюсь, я сказал ясно.



TerminateProcess — функция асинхронная, т. е. она сообщает системе, что Вы хотите завершить процесс, но к тому времени, когда она вернет управление, процесс может быть еще не уничтожен. Так что, если Вам нужно точно знать момент завершения процесса, используйте *WaitForSingleObject* (см. главу 9) или аналогичную функцию, передав ей описатель этого процесса.

Когда все потоки процесса уходят

В такой ситуации (а она может возникнуть, если все потоки вызвали *ExitThread* или их закрыли вызовом *TerminateThread*) операционная система больше не считает нужным «содержать» адресное пространство данного процесса. Обнаружив, что в процессе не исполняется ни один поток, она немедленно завершает его. При этом код завершения процесса приравнивается коду завершения последнего потока.

Что происходит при завершении процесса

А происходит вот что:

1. Выполнение всех потоков в процессе прекращается.
2. Все User- и GDI-объекты, созданные процессом, уничтожаются, а объекты ядра закрываются (если их не использует другой процесс).
3. Код завершения процесса меняется со значения `STILL_ACTIVE` на код, переданный в `ExitProcess` или `TerminateProcess`.
4. Объект ядра «процесс» переходит в свободное, или незанятое (`signaled`), состояние. (Подробнее на эту тему см. главу 9.) Прочие потоки в системе могут приступить к выполнению вплоть до завершения данного процесса.
5. Счетчик объекта ядра «процесс» уменьшается на 1.

Связанный с завершающим процессом объект ядра не высвобождается, пока не будут закрыты ссылки на него и из других процессов. В момент завершения процесса система автоматически уменьшает счетчик пользователей этого объекта на 1, и объект разрушается, как только его счетчик обнуляется. Кроме того, закрытие процесса не приводит к автоматическому завершению порожденных им процессов.

По завершении процесса его код и выделенные ему ресурсы удаляются из памяти. Однако область памяти, выделенная системой для объекта ядра «процесс», не освобождается, пока счетчик числа его пользователей не достигнет нуля. А это произойдет, когда все прочие процессы, создавшие или открывшие описатели для ныне-покойного процесса, уведомят систему (вызовом `CloseHandle`) о том, что ссылки на этот процесс им больше не нужны.

Описатели завершенного процесса уже мало на что пригодны. Разве что родительский процесс, вызвав функцию `GetExitCodeProcess`, может проверить, завершен ли процесс, идентифицируемый параметром `hProcess`, и, если да, определить код завершения:

```
BOOL GetExitCodeProcess(  
    HANDLE hProcess,  
    PDWORD pdwExitCode);
```

Код завершения возвращается как значение типа `DWORD`, на которое указывает `pdwExitCode`. Если на момент вызова `GetExitCodeProcess` процесс еще не завершился, в `DWORD` заносится идентификатор `STILL_ACTIVE` (определенный как `0x103`). А если он уничтожен, функция возвращает реальный код его завершения.

Вероятно, Вы подумали, что можно написать код, который, периодически вызывая функцию `GetExitCodeProcess` и проверяя возвращаемое ею значение, определял бы момент завершения процесса. В принципе такой код мог бы сработать во многих ситуациях, но он был бы неэффективен. Как правильно решить эту задачу, я расскажу в следующем разделе.

Дочерние процессы

При разработке приложения часто бывает нужно, чтобы какую-то операцию выполнил другой блок кода. Поэтому — хочешь, не хочешь — приходится постоянно вызывать функции или подпрограммы. Но вызов функции приводит к приостановке выполнения основного кода Вашей программы до возврата из вызванной функции. Альтернативный способ — передать выполнение какой-то операции другому потоку в пределах данного процесса (поток, разумеется, нужно сначала создать). Это позво-

лит основному коду программы продолжить работу в то время, как дополнительный поток будет выполнять другую операцию. Прием весьма удобный, но, когда основному потоку потребуется узнать результаты работы другого потока, Вам не избежать проблем, связанных с синхронизацией.

Есть еще один прием: Ваш процесс порождает дочерний и возлагает на него выполнение части операций. Будем считать, что эти операции очень сложны. Допустим, для их реализации Вы просто создаете новый поток внутри того же процесса. Вы пишете тот или иной код, тестируете его и получаете некорректный результат — может, ошиблись в алгоритме или запутались в ссылках и случайно перезаписали какие-нибудь важные данные в адресном пространстве своего процесса. Так вот, один из способов защитить адресное пространство основного процесса от подобных ошибок как раз и состоит в том, чтобы передать часть работы отдельному процессу. Далее можно или подождать, пока он завершится, или продолжить работу параллельно с ним.

К сожалению, дочернему процессу, по-видимому, придется оперировать с данными, содержащимися в адресном пространстве родительского процесса. Было бы не плохо, чтобы он работал исключительно в своем адресном пространстве, а в «Вашем» — просто считывал нужные ему данные; тогда он не сможет что-то испортить в адресном пространстве родительского процесса. В Windows предусмотрено несколько способов обмена данными между процессами: DDE (Dynamic Data Exchange), OLE, каналы (pipes), почтовые ящики (mailslots) и т. д. А один из самых удобных способов, обеспечивающих совместный доступ к данным, — использование файлов, проецируемых в память (*memory-mapped files*). (Подробнее на эту тему см. главу 17.)

Если Вы хотите создать новый процесс, заставить его выполнить какие-либо операции и дождаться их результатов, напишите примерно такой код:

```
PROCESS_INFORMATION pi;
DWORD dwExitCode;

// порождаем дочерний процесс
BOOL fSuccess = CreateProcess(..., &pi);
if (fSuccess) {

    // закрываем описатель потока, как только необходимость в нем отпадает!
    CloseHandle(pi.hThread);

    // приостанавливаем выполнение родительского процесса,
    // пока не завершится дочерний процесс
    WaitForSingleObject(pi.hProcess, INFINITE);

    // дочерний процесс завершился; получаем код его завершения
    GetExitCodeProcess(pi.hProcess, &dwExitCode);

    // закрываем описатель процесса, как только необходимость в нем отпадает!
    CloseHandle(pi.hProcess);
}
```

В этом фрагменте кода мы создали новый процесс и, если это прошло успешно, вызвали функцию *WaitForSingleObject*:

```
DWORD WaitForSingleObject(HANDLE hObject, DWORD dwTimeOut);
```

Подробное рассмотрение данной функции мы отложим до главы 9, а сейчас ограничимся одним соображением. Функция задерживает выполнение кода до тех пор,

пока объект, определяемый параметром *bObject*, не перейдет в свободное (незанятое) состояние. Объект «процесс» переходит в такое состояние при его завершении. Поэтому вызов *WaitForSingleObject* приостанавливает выполнение потока родительского процесса до завершения порожденного им процесса. Когда *WaitForSingleObject* вернет управление, Вы узнаете код завершения дочернего процесса через функцию *GetExitCodeProcess*.

Обращение к *CloseHandle* в приведенном выше фрагменте кода заставляет систему уменьшить значения счетчиков объектов «поток» и «процесс» до нуля и тем самым освободить память, занимаемую этими объектами.

Вы, наверное, заметили, что в этом фрагменте я закрыл описатель объекта ядра «первичный поток» (принадлежащий дочернему процессу) сразу после возврата из *CreateProcess*. Это не приводит к завершению первичного потока дочернего процесса — просто уменьшает счетчик, связанный с упомянутым объектом. А вот почему это делается — и, кстати, даже рекомендуется делать — именно так, станет ясно из простого примера. Допустим, первичный поток дочернего процесса порождает еще один поток, а сам после этого завершается. В этот момент система может высвободить объект «первичный поток» дочернего процесса из памяти, если у родительского процесса нет описателя данного объекта. Но если родительский процесс располагает таким описателем, система не сможет удалить этот объект из памяти до тех пор, пока и родительский процесс не закроет его описатель.

Запуск обособленных дочерних процессов

Что ни говори, но чаще приложение все-таки создает другие процессы как *обособленные* (*detached processes*). Это значит, что после создания и запуска нового процесса родительскому процессу нет нужды с ним взаимодействовать или ждать, пока тот закончит работу. Именно так и действует Explorer: запускает для пользователя новые процессы, а дальше его уже не волнует, что там с ними происходит.

Чтобы обрубить все пуповины, связывающие Explorer с дочерним процессом, ему нужно (вызовом *CloseHandle*) закрыть свои описатели, связанные с новым процессом и его первичным потоком. Приведенный ниже фрагмент кода демонстрирует, как, создав процесс, сделать его обособленным:

```
PROCESS_INFORMATION pi;  
  
BOOL fSuccess = CreateProcess(..., &pi);  
if (fSuccess) {  
    // разрешаем системе уничтожить объекты ядра "процесс" и "поток"  
    // сразу после завершения дочернего процесса  
    CloseHandle(pi.hThread);  
    CloseHandle(pi.hProcess);  
}
```

Перечисление процессов, выполняемых в системе

Многие разработчики программного обеспечения пытаются создавать инструментальные средства или утилиты для Windows, требующие перечисления процессов, выполняемых в системе. Изначально в Windows API не было функций, которые позволяли бы это делать. Однако в Windows NT ведется постоянно обновляемая база данных Performance Data. В ней содержится чуть ли не тонна информации, доступной через функции реестра вроде *RegQueryValueEx*, для которой надо указать корне-

вой раздел HKEY_PERFORMANCE_DATA. Мало кто из программистов знает об этой базе данных, и причины тому кроются, на мой взгляд, в следующем.

- Для нее не предусмотрено никаких специфических функций; нужно использовать обычные функции реестра.
- Ее нет в Windows 95 и Windows 98.
- Структура информации в этой базе данных очень сложна; многие просто избегают ею пользоваться (и другим не советуют).

Чтобы упростить работу с этой базой данных, Microsoft создала набор функций под общим названием Performance Data Helper (содержащийся в PDH.dll). Если Вас интересует более подробная информация о библиотеке PDH.dll, ищите раздел по функциям Performance Data Helper в документации Platform SDK.

Как я уже упоминал, в Windows 95 и Windows 98 такой базы данных нет. Вместо них предусмотрен набор функций, позволяющих перечислять процессы. Они включены в ToolHelp API. За информацией о них я вновь отсылаю Вас к документации Platform SDK — ищите разделы по функциям *Process32First* и *Process32Next*.

Но самое смешное, что разработчики Windows NT, которым ToolHelp-функции явно не нравятся, не включили их в Windows NT. Для перечисления процессов они создали свой набор функций под общим названием Process Status (содержащийся в PSAPI.dll). Так что ищите в документации Platform SDK раздел по функции *Emit-Processes*.

Microsoft, которая до сих пор, похоже, старалась усложнить жизнь разработчикам инструментальных средств и утилит, все же включила ToolHelp-функции в Windows 2000. Наконец-то эти разработчики смогут унифицировать свой код хотя бы для Windows 95, Windows 98 и Windows 2000!

Программа-пример ProcessInfo

Эта программа, «04 ProcessInfo.exe» (см. листинг на рис. 4-6), демонстрирует, как создать очень полезную утилиту на основе ToolHelp-функций. Файлы исходного кода и ресурсов программы находятся в каталоге 04-ProcessInfo на компакт-диске, прилагаемом к книге. После запуска ProcessInfo открывается окно, показанное на рис. 4-4.

ProcessInfo сначала перечисляет все процессы, выполняемые в системе, а затем выводит в верхний раскрывающийся список имена и идентификаторы каждого процесса. Далее выбирается первый процесс и информация о нем показывается в большом текстовом поле, доступном только для чтения. Как видите, для текущего процесса сообщается его идентификатор (вместе с идентификатором родительского процесса), класс приоритета и количество потоков, выполняемых в настоящий момент в контексте процесса. Объяснение большей части этой информации выходит за рамки данной главы, но будет рассматриваться в последующих главах.

При просмотре списка процессов становится доступен элемент меню VMMap. (Он отключается, когда Вы переключаетесь на просмотр информации о модулях.) Выбрав элемент меню VMMap, Вы запускаете программу-пример VMMap (см. главу 14). Эта программа «проходит» по адресному пространству выбранного процесса.

В информацию о модулях входит список всех модулей (EXE- и DLL-файлов), спроецированных на адресное пространство текущего процесса. Фиксированным модулем (fixed module) считается тот, который был явно загружен при инициализации процесса. Для явно загруженных DLL показываются счетчики числа пользователей этих DLL. Во втором столбце выводится базовый адрес памяти, на который спроеци-

рован модуль. Если модуль размещён не по заданному для него базовому адресу, в скобках появляется и этот адрес. В третьем столбце сообщается размер модуля в байтах, а в последнем — полное (вместе с путём) имя файла этого модуля. И, наконец, внизу показывается информация о потоках, выполняемых в данный момент в контексте текущего процесса. При этом отображается идентификатор потока (thread ID, TID) и его приоритет.

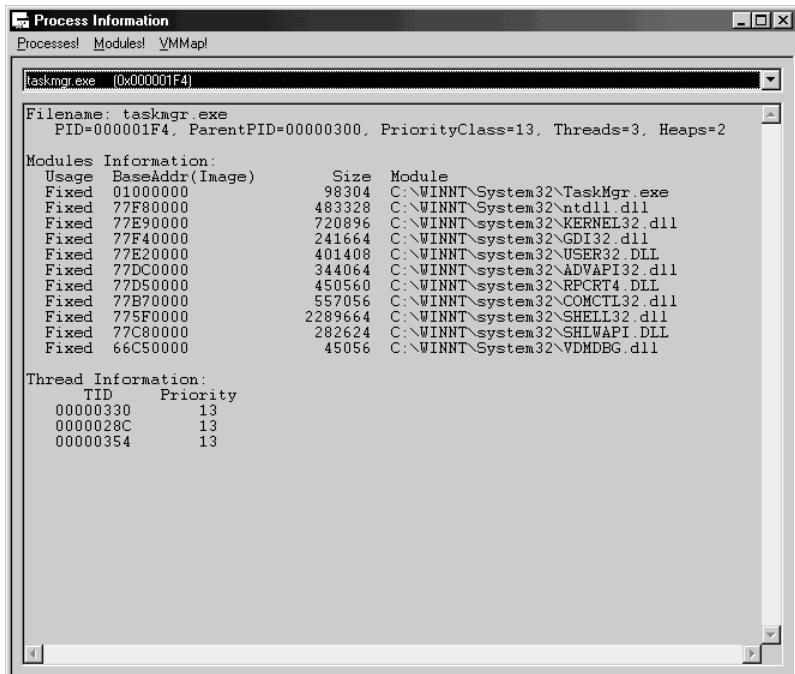


Рис. 4-4. *ProcessInfo* в действии

В дополнение к информации о процессах Вы можете выбрать элемент меню *Modules!*. Это заставит *ProcessInfo* перечислить все модули, загруженные в системе, и поместить их имена в верхний раскрывающийся список. Далее *ProcessInfo* выбирает первый модуль и выводит информацию о нем (рис. 4-5).

В этом режиме утилита *ProcessInfo* позволяет легко определить, в каких процессах задействован данный модуль. Как видите, полное имя модуля появляется в верхней части текстового поля, а в разделе *Process Information* перечисляются все процессы, содержащие этот модуль. Там же показываются идентификаторы и имена процессов, в которые загружен модуль, и его базовые адреса в этих процессах.

Всю эту информацию утилита *ProcessInfo* получает в основном от различных *ToolHelp*-функций. Чтобы чуточку упростить работу с *ToolHelp*-функциями, я создал C++-класс *CToolhelp* (содержащийся в файле *Toolhelp.h*). Он инкапсулирует все, что связано с получением «моментального снимка» состояния системы, и немного облегчает вызов других *ToolHelp*-функций.

Особый интерес представляет функция *GetModulePreferredBaseAddr* в файле *ProcessInfo.cpp*:

```
PVOID GetModulePreferredBaseAddr(
    DWORD dwProcessId,
    PVOID pvModuleRemote);
```

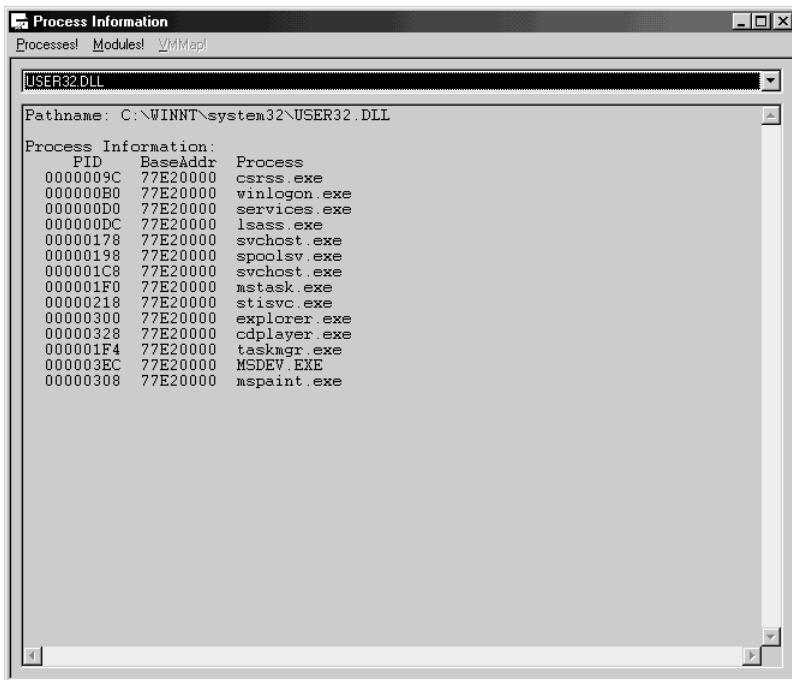


Рис. 4-5. *ProcessInfo* перечисляет все процессы, в адресные пространства которых загружен модуль *User32.dll*

Принимая идентификатор процесса и адрес модуля в этом процессе, она просматривает его адресное пространство, находит модуль и считывает информацию из заголовка модуля, чтобы определить, какой базовый адрес для него предпочтителен. Модуль должен всегда загружаться именно по этому адресу, а иначе приложения, использующие данный модуль, потребуют больше памяти и будут инициализироваться медленнее. Поскольку такая ситуация крайне нежелательна, моя утилита сообщает о случаях, когда модуль загружен не по предпочтительному базовому адресу. Впрочем, на эти темы мы поговорим в главе 20 (в разделе «Модификация базовых адресов модулей»).

ProcessInfo.cpp

```

/*
Модуль: ProcessInfo.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*/
#include "..\CmnHdr.h"      /* см. приложение A */
#include <windowsx.h>
#include <tlib32.h>
#include <tchar.h>
#include <stdarg.h>
#include <stdio.h>
#include "Toolhelp.h"
#include "Resource.h"

```

Рис. 4-6. Программа-пример *ProcessInfo*

см. след. стр.

Рис. 4-6. продолжение

```
//////////  
  
// добавляем строку в текстовое поле  
void AddText(HWND hwnd, PCTSTR pszFormat, ...) {  
  
    va_list argList;  
    va_start(argList, pszFormat);  
  
    TCHAR sz[20 * 1024];  
    Edit_GetText(hwnd, sz, chDIMOF(sz));  
    _vstprintf(_tcschr(sz, 0), pszFormat, argList);  
    Edit_SetText(hwnd, sz);  
    va_end(argList);  
}  
  
//////////  
  
VOID Dlg_PopulateProcessList(HWND hwnd) {  
  
    HWND hwndList = GetDlgItem(hwnd, IDC_PROCESSMODULELIST);  
    SetWindowRedraw(hwndList, FALSE);  
    ComboBox_ResetContent(hwndList);  
  
    CToolhelp thProcesses(TH32CS_SNAPPROCESS);  
    PROCESSENTRY32 pe = { sizeof(pe) };  
    BOOL fOk = thProcesses.ProcessFirst(&pe);  
    for (; fOk; fOk = thProcesses.ProcessNext(&pe)) {  
        TCHAR sz[1024];  
  
        // помещаем в список имя процесса (без пути) и идентификатор  
        PCTSTR pszExeFile = _tcsrchr(pe.szExeFile, TEXT('\\'));  
        if (pszExeFile == NULL) pszExeFile = pe.szExeFile;  
        else pszExeFile++; // пропускаем наклонную черту ("слэш")  
        wsprintf(sz, TEXT("%s (0x%08X)", pszExeFile, pe.th32ProcessID));  
        int n = ComboBox_AddString(hwndList, sz);  
  
        // сопоставляем идентификатор процесса с добавленным элементом  
        ComboBox_SetItemData(hwndList, n, pe.th32ProcessID);  
    }  
    ComboBox_SetCurSel(hwndList, 0); // выбираем первый элемент  
  
    // имитируем выбор пользователем первого элемента,  
    // чтобы в текстовом поле появилось что-нибудь интересное  
    FORWARD_WM_COMMAND(hwnd, IDC_PROCESSMODULELIST,  
        hwndList, CBN_SELCHANGE, SendMessage);  
  
    SetWindowRedraw(hwndList, TRUE);  
    InvalidateRect(hwndList, NULL, FALSE);  
}  
  
//////////
```

Рис. 4-6. продолжение

```

VOID Dlg_PopulateModuleList(HWND hwnd) {

    HWND hwndModuleHelp = GetDlgItem(hwnd, IDC_MODULEHELP);
    ListBox_ResetContent(hwndModuleHelp);
    CToolhelp thProcesses(TH32CS_SNAPPROCESS);
    PROCESSENTRY32 pe = { sizeof(pe) };
    BOOL fOk = thProcesses.ProcessFirst(&pe);
    for (; fOk; fOk = thProcesses.ProcessNext(&pe)) {

        CToolhelp thModules(TH32CS_SNAPMODULE, pe.th32ProcessID);
        MODULEENTRY32 me = { sizeof(me) };
        BOOL fOk = thModules.ModuleFirst(&me);
        for (; fOk; fOk = thModules.ModuleNext(&me)) {
            int n = ListBox_FindStringExact(hwndModuleHelp, -1, me.szExePath);
            if (n == LB_ERR) {
                // этот модуль еще не был добавлен
                ListBox_AddString(hwndModuleHelp, me.szExePath);
            }
        }
    }

    HWND hwndList = GetDlgItem(hwnd, IDC_PROCESSMODULELIST);
    SetWindowRedraw(hwndList, FALSE);
    ComboBox_ResetContent(hwndList);
    int nNumModules = ListBox_GetCount(hwndModuleHelp);
    for (int i = 0; i < nNumModules; i++) {
        TCHAR sz[1024];
        ListBox_GetText(hwndModuleHelp, i, sz);
        // помещаем в список имя модуля (без пути)
        int nIndex = ComboBox_AddString(hwndList, _tcsrchr(sz, TEXT('\\')) + 1);
        // сопоставляем индекс полного пути с добавленным элементом
        ComboBox_SetItemData(hwndList, nIndex, i);
    }

    ComboBox_SetCurSel(hwndList, 0); // выбираем первый элемент

    // имитируем выбор пользователем первого элемента,
    // чтобы в текстовом поле появилось что-нибудь интересное
    FORWARD_WM_COMMAND(hwnd, IDC_PROCESSMODULELIST,
        hwndList, CBN_SELCHANGE, SendMessage);

    SetWindowRedraw(hwndList, TRUE);
    InvalidateRect(hwndList, NULL, FALSE);
}

///////////////////////////////
PVVOID GetModulePreferredBaseAddr(DWORD dwProcessId, PVVOID pvModuleRemote) {

    PVVOID pvModulePreferredBaseAddr = NULL;
}

```

см. след. стр.

Рис. 4-6. продолжение

```
IMAGE_DOS_HEADER idh;
IMAGE_NT_HEADERS inth;

// считываем DOS-заголовок удаленного модуля
Toolhelp32ReadProcessMemory(dwProcessId,
    pvModuleRemote, &idh, sizeof(idh), NULL);

// проверяем DOS-заголовок его образа
if (idh.e_magic == IMAGE_DOS_SIGNATURE) {
    // считываем NT-заголовок удаленного модуля
    Toolhelp32ReadProcessMemory(dwProcessId,
        (PBYTE) pvModuleRemote + idh.e_lfanew, &inth, sizeof(inth), NULL);

    // проверяем NT-заголовок его образа
    if (inth.Signature == IMAGE_NT_SIGNATURE) {
        // NT-заголовок корректен,
        // получаем предпочтительный базовый адрес для данного образа
        pvModulePreferredBaseAddr = (PVOID) inth.OptionalHeader.ImageBase;
    }
}
return(pvModulePreferredBaseAddr);
}

////////// VOID ShowProcessInfo(HWND hwnd, DWORD dwProcessID) {

SetWindowText(hwnd, TEXT("")); // очищаем поле вывода
CToolhelp th(TH32CS_SNAPALL, dwProcessID);

// показываем подробную информацию о процессе
PROCESSENTRY32 pe = { sizeof(pe) };
BOOL fOk = th.ProcessFirst(&pe);
for (; fOk; fOk = th.ProcessNext(&pe)) {
    if (pe.th32ProcessID == dwProcessID) {
        AddText(hwnd, TEXT("Filename: %s\r\n"), pe.szExeFile);
        AddText(hwnd, TEXT(" PID=%08X, ParentPID=%08X, ")
            TEXT("PriorityClass=%d, Threads=%d, Heaps=%d\r\n"),
            pe.th32ProcessID, pe.th32ParentProcessID,
            pe.pcPriClassBase, pe.cntThreads,
            th.HowManyHeaps());
        break; // продолжать цикл больше не нужно
    }
}

// показываем модули в процессе;
// подсчитываем количество символов для вывода адреса
const int cchAddress = sizeof(PVOID) * 2;
AddText(hwnd, TEXT("\r\nModules Information:\r\n"))
    TEXT(" Usage %-s(%-*s) %8s Module\r\n"),
    cchAddress, TEXT("BaseAddr"));
```

Рис. 4-6. продолжение

```

cchAddress, TEXT("ImagAddr"), TEXT("Size"));

MODULEENTRY32 me = { sizeof(me) };
f0k = th.ModuleFirst(&me);

for (; f0k; f0k = th.ModuleNext(&me)) {
    if (me.ProccntUsage == 65535) {
        // модуль загружен неявно, и его нельзя выгрузить
        AddText(hwnd, TEXT(" Fixed"));
    } else {
        AddText(hwnd, TEXT("%5d"), me.ProccntUsage);
    }
    PVOID pvPreferredBaseAddr =
        GetModulePreferredBaseAddr(pe.th32ProcessID, me.modBaseAddr);
    if (me.modBaseAddr == pvPreferredBaseAddr) {
        AddText(hwnd, TEXT("%p %*s %8u %s\r\n"),
            me.modBaseAddr, cchAddress, TEXT(""),
            me.modBaseSize, me.szExePath);
    } else {
        AddText(hwnd, TEXT("%p(%p) %8u %s\r\n"),
            me.modBaseAddr, pvPreferredBaseAddr, me.modBaseSize, me.szExePath);
    }
}

// показываем потоки в процессе
AddText(hwnd, TEXT("\r\nThread Information:\r\n"))
    TEXT("      TID      Priority\r\n"));
THREADENTRY32 te = { sizeof(te) };
f0k = th.ThreadFirst(&te);
for (; f0k; f0k = th.ThreadNext(&te)) {
    if (te.th32OwnerProcessID == dwProcessID) {
        int nPriority = te.tpBasePri + te.tpDeltaPri;
        if ((te.tpBasePri < 16) && (nPriority > 15)) nPriority = 15;
        if ((te.tpBasePri > 15) && (nPriority > 31)) nPriority = 31;
        if ((te.tpBasePri < 16) && (nPriority < 1)) nPriority = 1;
        if ((te.tpBasePri > 15) && (nPriority < 16)) nPriority = 16;
        AddText(hwnd, TEXT("%08X      %2d\r\n"),
            te.th32ThreadID, nPriority);
    }
}
////////////////////////////////////////////////////////////////

VOID ShowModuleInfo(HWND hwnd, LPCTSTR pszModulePath) {

    SetWindowText(hwnd, TEXT("")); // очищаем поле вывода

    CToolhelp thProcesses(TH32CS_SNAPPROCESS);
    PROCESSENTRY32 pe = { sizeof(pe) };
    BOOL f0k = thProcesses.ProcessFirst(&pe);

```

см. след. стр.

Рис. 4-6. продолжение

```
AddText(hwnd, TEXT("Pathname: %s\r\nr\n"), pszModulePath);
AddText(hwnd, TEXT("Process Information:\r\n"));
AddText(hwnd, TEXT("      PID      BaseAddr Process\r\n"));
for ( ; f0k; f0k = thProcesses.ProcessNext(&pe)) {
    CToolhelp thModules(TH32CS_SNAPMODULE, pe.th32ProcessID);
    MODULEENTRY32 me = { sizeof(me) };
    BOOL f0k = thModules.ModuleFirst(&me);
    for ( ; f0k; f0k = thModules.ModuleNext(&me)) {
        if (_tcscmp(me.szExePath, pszModulePath) == 0) {
            AddText(hwnd, TEXT(" %08X %p %s\r\n"),
                    pe.th32ProcessID, me.modBaseAddr, pe.szExeFile);
        }
    }
}
////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {
    chSETDLGICONS(hwnd, IDI_PROCESSINFO);

    // убираем окно списка модулей
    ShowWindow(GetDlgItem(hwnd, IDC_MODULEHELP), SW_HIDE);

    // пусть в окне результатов используется фиксированный шрифт
    SetWindowFont(GetDlgItem(hwnd, IDC_RESULTS),
                  GetStockFont(ANSI_FIXED_FONT), FALSE);

    // по умолчанию показываем список выполняемых процессов
    Dlg_PopulateProcessList(hwnd);

    return(TRUE);
}
////////////////////////////////////////////////////////////////

BOOL Dlg_OnSize(HWND hwnd, UINT state, int cx, int cy) {
    RECT rc;
    int n = LOWORD(GetDialogBaseUnits());

    HWND hwndCtl = GetDlgItem(hwnd, IDC_PROCESSMODULELIST);
    GetClientRect(hwndCtl, &rc);
    SetWindowPos(hwndCtl, NULL, n, n, cx - n - n, rc.bottom, SWP_NOZORDER);

    hwndCtl = GetDlgItem(hwnd, IDC_RESULTS);
    SetWindowPos(hwndCtl, NULL, n, n + rc.bottom + n,
                 cx - n - n, cy - (n + rc.bottom + n) - n, SWP_NOZORDER);

    return(0);
}
```

Рис. 4-6. продолжение

```
//////////  

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {  

    static BOOL s_fProcesses = TRUE;  

    switch (id) {  

        case IDCANCEL:  

            EndDialog(hwnd, id);  

            break;  

        case ID_PROCESSES:  

            s_fProcesses = TRUE;  

            EnableMenuItem(GetMenu(hwnd), ID_VMMAP, MF_BYCOMMAND | MF_ENABLED);  

            DrawMenuBar(hwnd);  

            Dlg_PopulateProcessList(hwnd);  

            break;  

        case ID_MODULES:  

            EnableMenuItem(GetMenu(hwnd), ID_VMMAP, MF_BYCOMMAND | MF_GRAYED);  

            DrawMenuBar(hwnd);  

            s_fProcesses = FALSE;  

            Dlg_PopulateModuleList(hwnd);  

            break;  

        case IDC_PROCESSMODULELIST:  

            if (codeNotify == CBN_SELCHANGE) {  

                DWORD dw = ComboBox_GetCurSel(hwndCtl);  

                if (s_fProcesses) {  

                    dw = (DWORD) ComboBox_GetItemData(hwndCtl, dw); // ID процесса  

                    ShowProcessInfo(GetDlgItem(hwnd, IDC_RESULTS), dw);  

                } else {  

                    // индекс в окне вспомогательного списка  

                    dw = (DWORD) ComboBox_GetItemData(hwndCtl, dw);  

                    TCHAR szModulePath[1024];  

                    ListBox_GetText(GetDlgItem(hwnd, IDC_MODULEHELP), dw, szModulePath);  

                    ShowModuleInfo(GetDlgItem(hwnd, IDC_RESULTS), szModulePath);  

                }  

            }  

            break;  

        case ID_VMMAP:  

            STARTUPINFO si = { sizeof(si) };  

            PROCESS_INFORMATION pi;  

            TCHAR szCmdLine[1024];  

            HWND hwndCB = GetDlgItem(hwnd, IDC_PROCESSMODULELIST);  

            DWORD dwProcessId = (DWORD)  

                ComboBox_GetItemData(hwndCB, ComboBox_GetCurSel(hwndCB));  

            wsprintf(szCmdLine, TEXT("\\"14 VMMAP\" %d"), dwProcessId);  

            BOOL fOk = CreateProcess(NULL, szCmdLine, NULL, NULL,  

                FALSE, 0, NULL, NULL, &si, &pi);  

    }  

}
```

см. след. стр.

Рис. 4-6. продолжение

```
if (fOk) {
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
} else {
    chMB("Failed to execute VMMAP.EXE.");
}
break;
}

///////////////////////////////



INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

switch (uMsg) {
    chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
    chHANDLE_DLGMMSG(hwnd, WM_SIZE,           Dlg_OnSize);
    chHANDLE_DLGMMSG(hwnd, WM_COMMAND,        Dlg_OnCommand);
}
return(FALSE);
}

///////////////////////////////



int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

CToolhelp::EnableDebugPrivilege(TRUE);
DialogBox(hinstExe, MAKEINTRESOURCE(IDD_PROCESSINFO), NULL, Dlg_Proc);
CToolhelp::EnableDebugPrivilege(FALSE);
return(0);
}

/////////////////////////////// Конец файла ////////////////////////
```

Toolhelp.h

```
*****
Модуль: Toolhelp.h
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****



#include "..\CmnHdr.h"      /* см. приложение A */
#include <tlhelp32.h>
#include <tchar.h>

///////////////////////////////



class CToolhelp {
private:
    HANDLE m_hSnapshot;
```

Рис. 4-6. продолжение

```

public:
    CToolhelp(DWORD dwFlags = 0, DWORD dwProcessID = 0);
    ~CToolhelp();

    BOOL CreateSnapshot(DWORD dwFlags, DWORD dwProcessID = 0);

    BOOL ProcessFirst(PPROCESSENTRY32 ppe) const;
    BOOL ProcessNext(PPROCESSENTRY32 ppe) const;
    BOOL ProcessFind(DWORD dwProcessId, PPROCESSENTRY32 ppe) const;

    BOOL ModuleFirst(PMODULEENTRY32 pme) const;
    BOOL ModuleNext(PMODULEENTRY32 pme) const;
    BOOL ModuleFind(PVOID pvBaseAddr, PMODULEENTRY32 pme) const;
    BOOL ModuleFind(PTSTR pszModName, PMODULEENTRY32 pme) const;

    BOOL ThreadFirst(PTHREADENTRY32 pte) const;
    BOOL ThreadNext(PTHREADENTRY32 pte) const;

    BOOL HeapListFirst(PHEAPLIST32 phl) const;
    BOOL HeapListNext(PHEAPLIST32 phl) const;
    int HowManyHeaps() const;

    // Примечание: функции, оперирующие с блоками памяти в куче, не ссылаются
    // на "снимок", а просто каждый раз с самого начала просматривают кучу
    // процесса. Если исследуемый процесс изменит свою кучу, когда любая
    // из этих функций будет перечислять блоки в его куче, возможно вхождение
    // в бесконечный цикл.

    BOOL HeapFirst(PHEAPENTRY32 phe, DWORD dwProcessID,
        UINT_PTR dwHeapID) const;
    BOOL HeapNext(PHEAPENTRY32 phe) const;
    int HowManyBlocksInHeap(DWORD dwProcessID, DWORD dwHeapId) const;
    BOOL IsAHeap(HANDLE hProcess, PVOID pvBlock, PDWORD pdwFlags) const;

public:
    static BOOL EnableDebugPrivilege(BOOL fEnable = TRUE);
    static BOOL ReadProcessMemory(DWORD dwProcessID, LPCVOID pvBaseAddress,
        PVOID pvBuffer, DWORD cbRead, PDWORD pdwNumberOfBytesRead = NULL);
};

////////////////////////////////////////////////////////////////

inline CToolhelp::CToolhelp(DWORD dwFlags, DWORD dwProcessID) {

    m_hSnapshot = INVALID_HANDLE_VALUE;
    CreateSnapshot(dwFlags, dwProcessID);
}

////////////////////////////////////////////////////////////////

inline CToolhelp::~CToolhelp() {

```

см. след. стр.

Рис. 4-6. продолжение

```
if (m_hSnapshot != INVALID_HANDLE_VALUE)
    CloseHandle(m_hSnapshot);
}

///////////////////////////////



inline CToolhelp::CreateSnapshot(DWORD dwFlags, DWORD dwProcessID) {

    if (m_hSnapshot != INVALID_HANDLE_VALUE)
        CloseHandle(m_hSnapshot);

    if (dwFlags == 0) {
        m_hSnapshot = INVALID_HANDLE_VALUE;
    } else {
        m_hSnapshot = CreateToolhelp32Snapshot(dwFlags, dwProcessID);
    }
    return(m_hSnapshot != INVALID_HANDLE_VALUE);
}

///////////////////////////////



inline BOOL CToolhelp::EnableDebugPrivilege(BOOL fEnable) {

    // передавая приложению полномочия отладчика, мы разрешаем ему
    // видеть информацию о сервисных приложениях
    BOOL fOk = FALSE; // предполагаем худшее
    HANDLE hToken;

    // пытаемся открыть маркер доступа (access token) для этого процесса
    if (OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES,
        &hToken)) {

        TOKEN_PRIVILEGES tp;
        tp.PrivilegeCount = 1;
        LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &tp.Privileges[0].Luid);
        tp.Privileges[0].Attributes = fEnable ? SE_PRIVILEGE_ENABLED : 0;
        AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(tp), NULL, NULL);
        fOk = (GetLastError() == ERROR_SUCCESS);
        CloseHandle(hToken);
    }
    return(fOk);
}

///////////////////////////////



inline BOOL CToolhelp::ReadProcessMemory(DWORD dwProcessID,
    LPCVOID pvBaseAddress, PVOID pvBuffer, DWORD cbRead,
    PDWORD pdwNumberOfBytesRead) {

    return(Toolhelp32ReadProcessMemory(dwProcessID, pvBaseAddress, pvBuffer,
        cbRead, pdwNumberOfBytesRead));
}
```

Рис. 4-6. продолжение

```
//////////  

inline BOOL CToolhelp::ProcessFirst(PPROCESSENTRY32 ppe) const {  

    BOOL f0k = Process32First(m_hSnapshot, ppe);  

    if (f0k && (ppe->th32ProcessID == 0))  

        f0k = ProcessNext(ppe); // удаляем "[System Process]" (PID = 0)  

    return(f0k);  

}  

inline BOOL CToolhelp::ProcessNext(PPROCESSENTRY32 ppe) const {  

    BOOL f0k = Process32Next(m_hSnapshot, ppe);  

    if (f0k && (ppe->th32ProcessID == 0))  

        f0k = ProcessNext(ppe); // удаляем "[System Process]" (PID = 0)  

    return(f0k);  

}  

inline BOOL CToolhelp::ProcessFind(DWORD dwProcessId, PPROCESSENTRY32 ppe) const {  

    BOOL fFound = FALSE;  

    for (BOOL f0k = ProcessFirst(ppe); f0k; f0k = ProcessNext(ppe)) {  

        fFound = (ppe->th32ProcessID == dwProcessId);  

        if (fFound) break;  

    }  

    return(fFound);  

}  

//////////  

inline BOOL CToolhelp::ModuleFirst(PMODULEENTRY32 pme) const {  

    return(Module32First(m_hSnapshot, pme));  

}  

inline BOOL CToolhelp::ModuleNext(PMODULEENTRY32 pme) const {  

    return(Module32Next(m_hSnapshot, pme));  

}  

inline BOOL CToolhelp::ModuleFind(PVOID pvBaseAddr, PMODULEENTRY32 pme) const {  

    BOOL fFound = FALSE;  

    for (BOOL f0k = ModuleFirst(pme); f0k; f0k = ModuleNext(pme)) {  

        fFound = (pme->modBaseAddr == pvBaseAddr);  

        if (fFound) break;  

    }  

    return(fFound);  

}
```

см. след. стр.

Рис. 4-6. продолжение

```
inline BOOL CToolhelp::ModuleFind(PTSTR pszModName, PMODULEENTRY32 pme) const {
    BOOL fFound = FALSE;
    for (BOOL f0k = ModuleFirst(pme); f0k; f0k = ModuleNext(pme)) {
        fFound = (lstrcmpi(pme->szModule, pszModName) == 0) ||
            (lstrcmpi(pme->szExePath, pszModName) == 0);
        if (fFound) break;
    }
    return(fFound);
}

///////////////////////////////
inline BOOL CToolhelp::ThreadFirst(PTHREADENTRY32 pte) const {
    return(Thread32First(m_hSnapshot, pte));
}

inline BOOL CToolhelp::ThreadNext(PTHREADENTRY32 pte) const {
    return(Thread32Next(m_hSnapshot, pte));
}

///////////////////////////////
inline int CToolhelp::HowManyHeaps() const {

    int nHowManyHeaps = 0;
    HEAPLIST32 hl = { sizeof(hl) };
    for (BOOL f0k = HeapListFirst(&hl); f0k; f0k = HeapListNext(&hl))
        nHowManyHeaps++;
    return(nHowManyHeaps);
}

inline int CToolhelp::HowManyBlocksInHeap(DWORD dwProcessID,
    DWORD dwHeapID) const {

    int nHowManyBlocksInHeap = 0;
    HEAPENTRY32 he = { sizeof(he) };
    BOOL f0k = HeapFirst(&he, dwProcessID, dwHeapID);
    for (; f0k; f0k = HeapNext(&he))
        nHowManyBlocksInHeap++;
    return(nHowManyBlocksInHeap);
}

inline BOOL CToolhelp::HeapListFirst(PHEAPLIST32 phl) const {

    return(Heap32ListFirst(m_hSnapshot, phl));
}

inline BOOL CToolhelp::HeapListNext(PHEAPLIST32 phl) const {
```

Рис. 4-6. продолжение

```
    return(Heap32ListNext(m_hSnapshot, ph1));  
}  
  
inline BOOL CToolhelp::HeapFirst(PHEAPENTRY32 phe, DWORD dwProcessID,  
    UINT_PTR dwHeapID) const {  
  
    return(Heap32First(phe, dwProcessID, dwHeapID));  
}  
  
inline BOOL CToolhelp::HeapNext(PHEAPENTRY32 phe) const {  
  
    return(Heap32Next(phe));  
}  
  
inline BOOL CToolhelp::IsAHeap(HANDLE hProcess, PVOID pvBlock,  
    PDWORD pdwFlags) const {  
  
    HEAPLIST32 hl = { sizeof(hl) };  
    for (BOOL f0kHL = HeapListFirst(&hl); f0kHL; f0kHL = HeapListNext(&hl)) {  
        HEAPENTRY32 he = { sizeof(he) };  
        BOOL f0kHE = HeapFirst(&he, hl.th32ProcessID, hl.th32HeapID);  
        for (; f0kHE; f0kHE = HeapNext(&he)) {  
            MEMORY_BASIC_INFORMATION mbi;  
            VirtualQueryEx(hProcess, (PVOID) he.dwAddress, &mbi, sizeof(mbi));  
            if (chINRANGE(mbi.AllocationBase, pvBlock,  
                (PBYTE) mbi.AllocationBase + mbi.RegionSize)) {  
  
                *pdwFlags = hl.dwFlags;  
                return(TRUE);  
            }  
        }  
    }  
    return(FALSE);  
}  
  
/////////////////////////////// Конец файла ///////////////////////////////
```

Задания

Группу процессов зачастую нужно рассматривать как единую сущность. Например, когда Вы командуете Microsoft Developer Studio собрать проект, он порождает процесс Cl.exe, а тот в свою очередь может создать другие процессы (скажем, для дополнительных проходов компилятора). Но, если Вы пожелаете прервать сборку, Developer Studio должен каким-то образом завершить Cl.exe и все его дочерние процессы. Решение этой простой (и распространенной) проблемы в Windows было весьма затруднительно, поскольку она не отслеживает родственные связи между процессами. В частности, выполнение дочерних процессов продолжается даже после завершения родительского.

При разработке сервера тоже бывает полезно группировать процессы. Допустим, клиентская программа просит сервер выполнить приложение (которое создает ряд дочерних процессов) и сообщить результаты. Поскольку к серверу может обратиться сразу несколько клиентов, было бы неплохо, если бы он умел как-то ограничивать ресурсы, выделяемые каждому клиенту, и тем самым не давал бы одному клиенту монопольно использовать все серверные ресурсы. Под ограничения могли бы подпадать такие ресурсы, как процессорное время, выделяемое на обработку клиентского запроса, и размеры рабочего набора (working set). Кроме того, у клиентской программы не должно быть возможности завершить работу сервера и т. д.

В Windows 2000 введен новый объект ядра — задание (job). Он позволяет группировать процессы и помещать их в нечто вроде песочницы, которая определенным образом ограничивает их действия. Относитесь к этому объекту как к контейнеру процессов. Кстати, очень полезно создавать задание и с одним процессом — это позволяет налагать на процесс ограничения, которые иначе указать нельзя.

Взгляните на мою функцию *StartRestrictedProcess* (рис. 5-1). Она включает процесс в задание, которое ограничивает возможность выполнения определенных операций.

WINDOWS 98 Windows 98 не поддерживает задания.

```
void StartRestrictedProcess() {
    // создаем объект ядра "задание"
    HANDLE hjob = CreateJobObject(NULL, NULL);

    // вводим ограничения для процессов в задании

    // сначала определяем некоторые базовые ограничения
    JOBOBJECT_BASIC_LIMIT_INFORMATION jobli = { 0 };
```

Рис. 5-1. Функция *StartRestrictedProcess*

Рис. 5-1. продолжение

```

// процесс всегда выполняется с классом приоритета idle
jobli.PriorityClass = IDLE_PRIORITY_CLASS;

// задание не может использовать более одной секунды процессорного времени
jobli.PerJobUserTimeLimit.QuadPart = 10000000; // 1 секунда, выраженная в
                                                // 100-наносекундных интервалах

// два ограничения, которые я налагаю на задание (процесс)
jobli.LimitFlags = JOB_OBJECT_LIMIT_PRIORITY_CLASS
    | JOB_OBJECT_LIMIT_JOB_TIME;
SetInformationJobObject(hjob, JobObjectBasicLimitInformation, &jobli,
    sizeof(jobli));

// теперь вводим некоторые ограничения по пользовательскому интерфейсу
JOBOBJECT_BASIC_UI_RESTRICTIONS jobuir;
jobuir.UIRestrictionsClass = JOB_OBJECT_UILIMIT_NONE; // "замысловатый" нуль

// процесс не имеет права останавливать систему
jobuir.UIRestrictionsClass |= JOB_OBJECT_UILIMIT_EXITWINDOWS;

// процесс не имеет права обращаться к USER-объектам в системе
// (например, к другим окнам)
jobuir.UIRestrictionsClass |= JOB_OBJECT_UILIMIT_HANDLES;

SetInformationJobObject(hjob, JobObjectBasicUIRestrictions, &jobuir,
    sizeof(jobuir));

// Порождаем процесс, который будет размещен в задании.
// ПРИМЕЧАНИЕ: процесс нужно сначала создать и только потом поместить
// в задание. А это значит, что поток процесса должен быть создан
// и тут же приостановлен, чтобы он не смог выполнить какой-нибудь код
// еще до введения ограничений.
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
CreateProcess(NULL, "CMD", NULL, NULL, FALSE,
    CREATE_SUSPENDED, NULL, NULL, &si, &pi);
// Включаем процесс в задание.
// ПРИМЕЧАНИЕ: дочерние процессы, порождаемые этим процессом,
// автоматически становятся частью того же задания.
AssignProcessToJobObject(hjob, pi.hProcess);

// теперь потоки дочерних процессов могут выполнять код
ResumeThread(pi.hThread);
CloseHandle(pi.hThread);

// ждем, когда процесс завершится или будет исчерпан
// лимит процессорного времени, указанный для задания
HANDLE h[2];
h[0] = pi.hProcess;
h[1] = hjob;

```

см. след. стр.

Рис. 5-1. продолжение

```
DWORD dw = WaitForMultipleObjects(2, h, FALSE, INFINITE);
switch (dw - WAIT_OBJECT_0) {
    case 0:
        // процесс завершился...
        break;
    case 1:
        // лимит процессорного времени исчерпан...
        break;
}
// проводим очистку
CloseHandle(pi.hProcess);
CloseHandle(hjob);
```

А теперь я объясню, как работает *StartRestrictedProcess*. Сначала я создаю новый объект ядра «задание», вызывая:

```
HANDLE CreateJobObject(
    PSECURITY_ATTRIBUTES psa,
    PCTSTR pszName);
```

Как и любая функция, создающая объекты ядра, *CreateJobObject* принимает в первом параметре информацию о защите и сообщает системе, должна ли она вернуть наследуемый описатель. Параметр *pszName* позволяет присвоить заданию имя, чтобы к нему могли обращаться другие процессы через функцию *OpenJobObject*.

```
HANDLE OpenJobObject(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);
```

Закончив работу с объектом-заданием, закройте его описатель, вызвав, как всегда, *CloseHandle*. Именно так я и делаю в конце своей функции *StartRestrictedProcess*. Имейте в виду, что закрытие объекта-задания не приводит к автоматическому завершению всех его процессов. На самом деле этот объект просто помечается как подлежащий разрушению, и система уничтожает его только после завершения всех включенных в него процессов.

Заметьте, что после закрытия описателя объект-задание становится недоступным для процессов, даже несмотря на то что объект все еще существует. Этот факт иллюстрирует следующий код:

```
// создаем именованный объект-задание
HANDLE hjob = CreateJobObject(NULL, TEXT("Jeff"));

// включаем в него наш процесс
AssignProcessToJobObject(hjob, GetCurrentProcess());

// закрытие объекта-задания не убивает ни наш процесс, ни само задание,
// но присвоенное ему имя ("Jeff") моментально удаляется
CloseHandle(hjob);

// пробуем открыть существующее задание
hjob = OpenJobObject(JOB_OBJECT_ALL_ACCESS, FALSE, TEXT("Jeff"));
// OpenJobObject терпит неудачу и возвращает NULL, поскольку имя ("Jeff")
```

```
// уже не указывает на объект-задание после вызова CloseHandle;
// получить описатель этого объекта больше нельзя
```

Определение ограничений, налагаемых на процессы в задании

Создав задание, Вы обычно строите «песочницу» (набор ограничений) для включаемых в него процессов. Ограничения бывают нескольких видов:

- базовые и расширенные базовые ограничения — не дают процессам в задании монопольно захватывать системные ресурсы;
- базовые ограничения по пользовательскому интерфейсу (UI) — блокируют возможность его изменения;
- ограничения, связанные с защитой, — перекрывают процессам в задании доступ к защищенным ресурсам (файлам, подразделам реестра и т. д.).

Ограничения на задание вводятся вызовом:

```
BOOL SetInformationJobObject(
    HANDLE hJob,
    JOBOBJECTINFOCLASS JobObjectInformationClass,
    PVOID pJobObjectInformation,
    DWORD cbJobObjectInformationLength);
```

Первый параметр определяет нужное Вам задание, второй параметр (перечисленного типа) — вид ограничений, третий — адрес структуры данных, содержащей подробную информацию о задаваемых ограничениях, а четвертый — размер этой структуры (используется для указания версии). Следующая таблица показывает, как устанавливаются ограничения.

Вид ограничений	Значение второго параметра	Структура, указываемая в третьем параметре
Базовые ограничения	<i>JobObjectBasicLimitInformation</i>	<code>JOBOBJECT_BASIC_LIMIT_INFORMATION</code>
Расширенные базовые ограничения	<i>JobObjectExtendedLimitInformation</i>	<code>JOBOBJECT_EXTENDED_LIMIT_INFORMATION</code>
Базовые ограничения по пользовательскому интерфейсу	<i>JobObjectBasicUIRestrictions</i>	<code>JOBOBJECT_BASIC_UI_RESTRICTIONS</code>
Ограничения, связанные с защитой	<i>JobObjectSecurityLimitInformation</i>	<code>JOBOBJECT_SECURITY_LIMIT_INFORMATION</code>

В функции *StartRestrictedProcess* я устанавливаю для задания лишь несколько базовых ограничений. Для этого я создаю структуру `JOB_OBJECT_BASIC_LIMIT_INFORMATION`, инициализирую ее и вызываю функцию *SetInformationJobObject*. Данная структура выглядит так:

```
typedef struct _JOBOBJECT_BASIC_LIMIT_INFORMATION {
    LARGE_INTEGER PerProcessUserTimeLimit;
    LARGE_INTEGER PerJobUserTimeLimit;
    DWORD        LimitFlags;
    DWORD        MinimumWorkingSetSize;
```

см. след. стр.

```
    DWORD      MaximumWorkingSetSize;
    DWORD      ActiveProcessLimit;
    DWORD_PTR   Affinity;
    DWORD      PriorityClass;
    DWORD      SchedulingClass;
} JOBOBJECT_BASIC_LIMIT_INFORMATION, *PJOBOBJECT_BASIC_LIMIT_INFORMATION;
```

Все элементы этой структуры кратко описаны в таблице 5-1.

Элементы	Описание	Примечание
<i>PerProcessUser-TimeLimit</i>	Максимальное время в пользовательском режиме, выделяемое каждому процессу (в порциях по 100 нс)	Система автоматически завершает любой процесс, который пытается использовать больше отведенного времени. Это ограничение вводится флагом <i>JOB_OBJECT_LIMIT_PROCESS_TIME</i> в <i>LimitFlags</i> .
<i>PerJobUser-TimeLimit</i>	Максимальное время в пользовательском режиме для всех процессов в данном задании (в порциях по 100 нс)	По умолчанию система автоматически завершает все процессы, когда заканчивается это время. Данное значение можно изменять в процессе выполнения задания. Это ограничение вводится флагом <i>JOB_OBJECT_LIMIT_JOB_TIME</i> в <i>LimitFlags</i> .
<i>LimitFlags</i>	Виды ограничений для задания	См. раздел после таблицы.
<i>MinimumWorking-SetSize</i> и <i>MaximumWorking-SetSize</i>	Верхний и нижний предел рабочего набора для каждого процесса (а не для всех процессов в задании)	Обычно рабочий набор процесса может расширяться за стандартный предел; указав <i>MaximumWorking-SetSize</i> , Вы введете жесткое ограничение. Когда размер рабочего набора какого-либо процесса достигнет заданного предела, процесс начнет сбрасывать свои страницы на диск. Вызовы функции <i>SetProcessWorking-SetSize</i> этим процессом будут игнорироваться, если только он не обращается к ней для того, чтобы очистить свой рабочий набор. Это ограничение вводится флагом <i>JOB_OBJECT_LIMIT_WORKINGSET</i> в <i>LimitFlags</i> .
<i>ActiveProcessLimit</i>	Максимальное количество процессов, одновременно выполняемых в задании	Любая попытка обойти такое ограничение приведет к завершению нового процесса с ошибкой «not enough quota» («превышение квоты»). Это ограничение вводится флагом <i>JOB_OBJECT_LIMIT_ACTIVE_PROCESS</i> в <i>LimitFlags</i> .
<i>Affinity</i>	Подмножество процессоров, на которых можно выполнять процессы этого задания	Для индивидуальных процессов это ограничение можно еще больше детализировать. Вводится флагом <i>JOB_OBJECT_LIMIT_AFFINITY</i> в <i>LimitFlags</i> .

Таблица 5-1. Элементы структуры *JOBOBJECT_BASIC_LIMIT_INFORMATION*

продолжение

Элементы	Описание	Примечание
<i>PriorityClass</i>	Класс приоритета для всех процессов в задании	Вызванная процессом функция <i>SetPriorityClass</i> сообщает об успехе даже в том случае, если на самом деле она не выполнила свою задачу, а <i>GetPriorityClass</i> возвращает класс приоритета, каковой и пытался установить процесс, хотя в реальности его класс может быть совсем другим. Кроме того, <i>SetThreadPriority</i> не может поднять приоритет потоков выше <i>normal</i> , но позволяет понизить его. Это ограничение вводится флагом <i>JOB_OBJECT_LIMIT_PRIORITY_CLASS</i> в <i>LimitFlags</i> .
<i>SchedulingClass</i>	Относительная продолжительность кванта времени, выделяемого всем потокам в задании	Этот элемент может принимать значения от 0 до 9; по умолчанию устанавливается 5. Подробнее о его назначении см. ниже. Это ограничение вводится флагом <i>JOB_OBJECT_LIMIT_SCHEDULING_CLASS</i> в <i>LimitFlags</i> .

Хочу пояснить некоторые вещи, связанные с этой структурой, которые, по-моему довольно туманно изложены в документации Platform SDK. Указывая ограничения для задания, Вы устанавливаете те или иные биты в элементе *LimitFlags*. Например, в *StartRestrictedProcess* я использовал флаги *JOB_OBJECT_LIMIT_PRIORITY_CLASS* и *JOB_OBJECT_LIMIT_JOB_TIME*, т. е. определил всего два ограничения.

При выполнении задание ведет учет по нескольким показателям — например, сколько процессорного времени уже использовали его процессы. Всякий раз, когда Вы устанавливаете базовые ограничения с помощью флага *JOB_OBJECT_LIMIT_JOB_TIME*, из общего процессорного времени, израсходованного всеми процессами, вычитается то, которое использовали завершившиеся процессы. Этот показатель сообщает, сколько процессорного времени израсходовали активные на данный момент процессы. А что если Вам понадобится изменить ограничения на доступ к подмножеству процессоров, не сбрасывая при этом учетную информацию по процессорному времени? Для этого Вы должны ввести новое базовое ограничение флагом *JOB_OBJECT_LIMIT_AFFINITY* и отказаться от флага *JOB_OBJECT_LIMIT_JOB_TIME*. Но тогда получится, что Вы снимаете ограничения на процессорное время.

Вы хотели другого: ограничить доступ к подмножеству процессоров, сохранив существующее ограничение на процессорное время, и не вычитать время, израсходованное завершенными процессами, из общего времени. Чтобы решить эту проблему, используйте специальный флаг *JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME*. Этот флаг и *JOB_OBJECT_LIMIT_JOB_TIME* являются взаимоисключающими. Флаг *JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME* указывает системе изменить ограничения, не вычитая процессорное время, использованное уже завершенными процессами.

Обсудим также элемент *SchedulingClass* структуры *JOBOBJECT_BASIC_LIMIT_INFORMATION*. Представьте, что для двух заданий определен класс приоритета *NORMAL_PRIORITY_CLASS*, а Вы хотите, чтобы процессы одного задания получали больше процессорного времени, чем процессы другого. Так вот, элемент *SchedulingClass* позволяет изменять распределение процессорного времени между заданиями с одинаковым

классом приоритета. Вы можете присвоить ему любое значение в пределах 0–9 (по умолчанию он равен 5). Увеличивая его значение, Вы заставляете Windows 2000 выделять потокам в процессах конкретного задания более длительный квант времени, а снижая — напротив, уменьшаете этот квант.

Допустим, у меня есть два задания с обычным (normal) классом приоритета: в каждом задании — по одному процессу, а в каждом процессе — по одному потоку (тоже с обычным приоритетом). В нормальной ситуации эти два потока обрабатывались бы процессором по принципу карусели и получали бы равные кванты процессорного времени. Но если я запишу в элемент *SchedulingClass* для первого задания значение 3, система будет выделять его потокам более короткий квант процессорного времени, чем потокам второго задания.

Используя *SchedulingClass*, избегайте слишком больших его значений, иначе Вы замедлите общую реакцию других заданий, процессов и потоков на какие-либо события в системе. Кроме того, учтите, что все сказанное здесь относится только к Windows 2000. В будущих версиях Windows планировщик потоков предполагается существенно изменить, чтобы операционная система могла более гибко планировать потоки в заданиях и процессах.

И последнее ограничение, которое заслуживает отдельного упоминания, связано с флагом *JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION*. Он отключает для всех процессов в задании вывод диалогового окна с сообщением о необработанном исключении. Система реагирует на этот флаг вызовом *SetErrorMode* с флагом *SEM_NOGPFAULTERRORBOX* для каждого из процессов в задании. Процесс, в котором возникнет необрабатываемое им исключение, немедленно завершается без уведомления пользователя. Этот флаг полезен в сервисных и других пакетных заданиях. В его отсутствие один из процессов в задании мог бы вызвать исключение и не завершиться, впустую расходуя системные ресурсы.

Помимо базовых ограничений, Вы можете устанавливать расширенные, для чего применяется структура *JOBOBJECT_EXTENDED_LIMIT_INFORMATION*:

```
typedef struct _JOBOBJECT_EXTENDED_LIMIT_INFORMATION {
    JOBOBJECT_BASIC_LIMIT_INFORMATION BasicLimitInformation;
    IO_COUNTERS IoInfo;
    SIZE_T ProcessMemoryLimit;
    SIZE_T JobMemoryLimit;
    SIZE_T PeakProcessMemoryUsed;
    SIZE_T PeakJobMemoryUsed;
} JOBOBJECT_EXTENDED_LIMIT_INFORMATION, *PJOBOBJECT_EXTENDED_LIMIT_INFORMATION;
```

Как видите, она включает структуру *JOBOBJECT_BASIC_LIMIT_INFORMATION*, являясь фактически ее надстройкой. Это несколько странная структура, потому что в ней есть элементы, не имеющие никакого отношения к определению ограничений для задания. Во-первых, элемент *IoInfo* зарезервирован, и Вы ни в коем случае не должны обращаться к нему. О том, как узнать значение счетчика ввода-вывода, я расскажу позже. Кроме того, элементы *PeakProcessMemoryUsed* и *PeakJobMemoryUsed* предназначены только для чтения и сообщают о максимальном объеме памяти, переданной соответственно одному из процессов или всем процессам в задании.

Остальные два элемента, *ProcessMemoryLimit* и *JobMemoryLimit*, ограничивают соответственно объем переданной памяти, который может быть использован одним из процессов или всеми процессами в задании. Чтобы задать любое из этих ограничений, укажите в элементе *LimitFlags* флаг *JOB_OBJECT_LIMIT_JOB_MEMORY* или *JOB_OBJECT_LIMIT_PROCESS_MEMORY*.

А теперь вернемся к прочим ограничениям, которые можно налагать на задания. Структура JOBOBJECT_BASIC_UI_RESTRICTIONS выглядит так:

```
typedef struct _JOBOBJECT_BASIC_UI_RESTRICTIONS {
    DWORD UIRestrictionsClass;
} JOBOBJECT_BASIC_UI_RESTRICTIONS, *PJOBOBJECT_BASIC_UI_RESTRICTIONS;
```

В этой структуре всего один элемент, *UIRestrictionsClass*, который содержит набор битовых флагов, кратко описанных в таблице 5-2.

Флаг	Описание
JOB_OBJECT_UILIMIT_EXITWINDOWS	Запрещает выдачу команд из процессов на выход из системы, завершение ее работы, перезагрузку или выключение компьютера через функцию <i>ExitWindowsEx</i>
JOB_OBJECT_UILIMIT_READCLIPBOARD	Запрещает процессам чтение из буфера обмена
JOB_OBJECT_UILIMIT_WRITECLIPBOARD	Запрещает процессам стирание буфера обмена
JOB_OBJECT_UILIMIT_SYSTEMPARAMETERS	Запрещает процессам изменение системных параметров через <i>SystemParametersInfo</i>
JOB_OBJECT_UILIMIT_DISPLAYSETTINGS	Запрещает процессам изменение параметров экрана через <i>ChangeDisplaySettings</i>
JOB_OBJECT_UILIMIT_GLOBALATOMS	Предоставляет заданию отдельную глобальную таблицу атомарного доступа (global atom table) и разрешает его процессам пользоваться только этой таблицей
JOB_OBJECT_UILIMIT_DESKTOP	Запрещает процессам создание новых рабочих столов или переключение между ними через функции <i>CreateDesktop</i> или <i>SwitchDesktop</i>
JOB_OBJECT_UILIMIT_HANDLES	Запрещает процессам в задании использовать USER-объекты (например, HWND), созданные внешними по отношению к этому заданию процессами

Таблица 5-2. Битовые флаги базовых ограничений по пользовательскому интерфейсу для объекта-задания

Последний флаг, JOB_OBJECT_UILIMIT_HANDLES, представляет особый интерес: он запрещает процессам в задании обращаться к USER-объектам, созданным внешними по отношению к этому заданию процессами. Так, запустив утилиту Microsoft Spy++ из задания, Вы не обнаружите никаких окон, кроме тех, которые создаст сама Spy++. На рис. 5-2 показано окно Microsoft Spy++ с двумя открытыми дочерними MDI-окнами. Заметьте, что в левой секции (Threads 1) содержится список потоков в системе. Кажется, что лишь у одного из них, 000006AC SPYXX, есть дочерние окна. А все дело в том, что я запустил Microsoft Spy++ из задания и ограничил ему права на использование описателей USER-объектов. В том же окне сообщается о потоках MSDEV и EXPLORER, но никаких упоминаний о созданных ими окнах нет. Уверяю Вас, эти потоки наверняка создали какие-нибудь окна — просто Spy++ лишена возможности их видеть. В правой секции (Windows 3) утилиты Spy++ должна показывать иерархию окон на рабочем столе, но там нет ничего, кроме одного элемента — 00000000. (Это не настоящий элемент, но Spy++ была обязана поместить сюда хоть что-нибудь.)

Обратите внимание, что такие ограничения односторонни, т. е. внешние процессы все равно видят USER-объекты, которые созданы процессами, включенными в задание. Например, если запустить Notepad в задании, а Spy++ — вне его, последняя увидит окно Notepad, даже если для задания указан флаг `JOB_OBJECT_UILIMIT_HANDLES`. Кроме того, Spy++, запущенная в отдельном задании, все равно увидит это окно Notepad, если только для ее задания не установлен флаг `JOB_OBJECT_UILIMIT_HANDLES`.

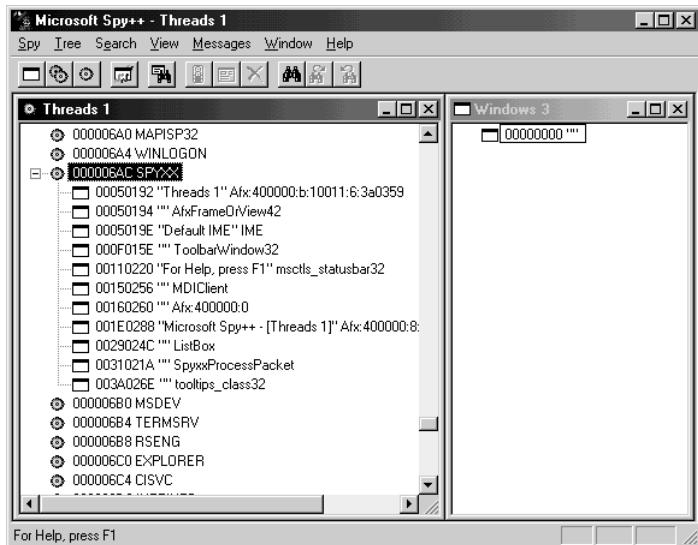


Рис. 5-2. Microsoft Spy++ работает в задании, которому ограничен доступ к описателям USER-объектов

Ограничение доступа к описателям USER-объектов — вещь изумительная, если Вы хотите создать по-настоящему безопасную песочницу, в которой будут «копаться» процессы Вашего задания. Однако часто бывает нужно, чтобы процесс в задании взаимодействовал с внешними процессами. Одно из самых простых решений здесь — использовать оконные сообщения, но, если процессам в задании доступ к описателям пользовательского интерфейса запрещен, ни один из них не сможет послать сообщение (синхронно или асинхронно) окну, созданному внешним процессом. К счастью, теперь есть функция, которая поможет решить эту проблему:

```
BOOL UserHandleGrantAccess(
    HANDLE hUserObj,
    HANDLE hjob,
    BOOL fGrant);
```

Параметр `hUserObj` идентифицирует конкретный USER-объект, доступ к которому Вы хотите предоставить или запретить процессам в задании. Это почти всегда описатель окна, но USER-объектом может быть, например, рабочий стол, программная ловушка, ярлык или меню. Последние два параметра, `hjob` и `fGrant`, указывают на задание и вид ограничения. Обратите внимание, что функция не сработает, если ее вызвать из процесса в том задании, на которое указывает `hjob`, — процесс не имеет права сам себе предоставлять доступ к объекту.

И последний вид ограничений, устанавливаемых для задания, относится к защите. (Введя в действие такие ограничения, Вы не сможете их отменить.) Структура `JOBOBJECT_SECURITY_LIMIT_INFORMATION` выглядит так:

```

typedef struct _JOBOBJECT_SECURITY_LIMIT_INFORMATION {
    DWORD SecurityLimitFlags;
    HANDLE JobToken;
    PTOKEN_GROUPS SidsToDelete;
    PTOKEN_PRIVILEGES PrivilegesToDelete;
    PTOKEN_GROUPS RestrictedSids;
} JOBOBJECT_SECURITY_LIMIT_INFORMATION, *PJOBOBJECT_SECURITY_LIMIT_INFORMATION;

```

Ее элементы описаны в следующей таблице.

Элемент	Описание
<i>SecurityLimitFlags</i>	Набор флагов, которые закрывают доступ администратору, запрещают маркер неограниченного доступа, принудительно назначают заданный маркер доступа, блокируют доступ по каким-либо идентификаторам защиты (security ID, SID) и отменяют указанные привилегии
<i>JobToken</i>	Маркер доступа, связываемый со всеми процессами в задании
<i>SidsToDelete</i>	Указывает, по каким SID не разрешается доступ
<i>PrivilegesToDelete</i>	Определяет привилегии, которые снимаются с маркера доступа
<i>RestrictedSids</i>	Задает набор SID, по которым запрещается доступ к любому защищенному объекту (deny-only SIDs); этот набор добавляется к маркеру доступа

Естественно, если Вы налагаете ограничения, то потом Вам, наверное, понадобится информация о них. Для этого вызовите:

```

BOOL QueryInformationJobObject(
    HANDLE hJob,
    JOBOBJECTINFOCLASS JobObjectInformationClass,
    PVOID pvJobObjectInformation,
    DWORD cbJobObjectInformationLength,
    PDWORD pdwReturnLength);

```

В эту функцию, как и в *SetInformationJobObject*, передается описатель задания, переменная перечислимого типа *JOBOBJECTINFOCLASS*. Она сообщает информацию об ограничениях, адрес и размер структуры данных, инициализируемой функцией. Последний параметр, *pdwReturnLength*, заполняется самой функцией и указывает, сколько байтов помещено в буфер. Если эти сведения Вас не интересуют (что обычно и бывает), передавайте в этом параметре NULL.



Процесс может получить информацию о своем задании, передав при вызове *QueryInformationJobObject* вместо описателя задания значение NULL. Это позволит ему выяснить установленные для него ограничения. Однако аналогичный вызов *SetInformationJobObject* даст ошибку, так как процесс не имеет права самостоятельно изменять заданные для него ограничения.

Включение процесса в задание

О'кэй, с ограничениями на этом закончим. Вернемся к *StartRestrictedProcess*. Установив ограничения для задания, я вызываю *CreateProcess* и создаю процесс, который помещаю в это задание. Я использую здесь флаг *CREATE_SUSPENDED*, и он приводит к тому, что процесс порождается, но код пока не выполняет. Поскольку *StartRestrictedProcess* вызывается из процесса, внешнего по отношению к заданию, его дочерний

процесс тоже не входит в это задание. Если бы я разрешил дочернему процессу немедленно начать выполнение кода, он проигнорировал бы мою песочницу со всеми ее ограничениями. Поэтому сразу после создания дочернего процесса и перед началом его работы я должен явно включить этот процесс в только что сформированное задание, вызвав:

```
BOOL AssignProcessToJobObject(  
    HANDLE hJob,  
    HANDLE hProcess);
```

Эта функция заставляет систему рассматривать процесс, идентифицируемый параметром *hProcess*, как часть существующего задания, на которое указывает *hJob*. Обратите внимание, что *AssignProcessToJobObject* позволяет включить в задание только тот процесс, который еще не относится ни к одному заданию. Как только процесс стал частью какого-нибудь задания, его нельзя переместить в другое задание или отпустить на волю. Кроме того, когда процесс, включенный в задание, порождает новый процесс, последний автоматически помещается в то же задание. Однако этот порядок можно изменить.

- Включая в *LimitFlags* структуры *JOBOBJECT_BASIC_LIMIT_INFORMATION* флаг *JOB_OBJECT_BREAKAWAY_OK*, Вы сообщаете системе, что новый процесс может выполняться вне задания. Потом Вы должны вызвать *CreateProcess* с новым флагом *CREATE_BREAKAWAY_FROM_JOB*. (Если Вы сделаете это без флага *JOB_OBJECT_BREAKAWAY_OK* в *LimitFlags*, функция *CreateProcess* завершится с ошибкой.) Такой механизм пригодится на случай, если новый процесс тоже управляет заданиями.
- Включая в *LimitFlags* структуры *JOBOBJECT_BASIC_LIMIT_INFORMATION* флаг *JOB_OBJECT_SILENT_BREAKAWAY_OK*, Вы тоже сообщаете системе, что новый процесс не является частью задания. Но указывать в *CreateProcess* какие-либо флаги на этот раз не потребуется. Данный механизм полезен для процессов, которым ничего не известно об объектах-заданиях.

Что касается *StartRestrictedProcess*, то после вызова *AssignProcessToJobObject* новый процесс становится частью задания. Далее язываю *ResumeThread*, чтобы поток нового процесса начал выполняться в рамках ограничений, установленных для задания. В этот момент я также закрываю описатель потока, поскольку он мне больше не нужен.

Завершение всех процессов в задании

Уверен, именно это Вы и будете делать чаще всего. В начале главы я упомянул о том, как непросто остановить сборку в Developer Studio, потому что для этого ему должны быть известны все процессы, которые успел создать его самый первый процесс. (Это очень каверзная задача. Как Developer Studio справляется с ней, я объяснял в своей колонке «Вопросы и ответы по Win32» в июньском выпуске Microsoft Systems Journal за 1998 год.) Подозреваю, что следующие версии Developer Studio будут использовать механизм заданий, и решать задачу, о которой мы с Вами говорили, станет гораздо легче.

Чтобы уничтожить все процессы в задании, Вы просто вызываете:

```
BOOL TerminateJobObject(  
    HANDLE hJob,  
    UINT uExitCode);
```

Вызов этой функции похож на вызов *TerminateProcess* для каждого процесса в задании и присвоение всем кодам завершения одного значения — *uExitCode*.

Получение статистической информации о задании

Мы уже обсудили, как с помощью *QueryInformationJobObject* получить информацию о текущих ограничениях, установленных для задания. Этой функцией можно пользоваться и для получения статистической информации. Например, чтобы выяснить базовые учетные сведения, вызовите ее, передав *JobObjectBasicAccountingInformation* во втором параметре и адрес структуры *JOBOBJECT_BASIC_ACCOUNTING_INFORMATION*:

```
typedef struct _JOBOBJECT_BASIC_ACCOUNTING_INFORMATION {
    LARGE_INTEGER TotalUserTime;
    LARGE_INTEGER TotalKernelTime;
    LARGE_INTEGER ThisPeriodTotalUserTime;
    LARGE_INTEGER ThisPeriodTotalKernelTime;
    DWORD TotalPageFaultCount;
    DWORD TotalProcesses;
    DWORD ActiveProcesses;
    DWORD TotalTerminatedProcesses;
} JOBOBJECT_BASIC_ACCOUNTING_INFORMATION, *PJOBOBJECT_BASIC_ACCOUNTING_INFORMATION;
```

Элементы этой структуры кратко описаны в таблице 5-3.

Элемент	Описание
<i>TotalUserTime</i>	Процессорное время, израсходованное процессами задания в пользовательском режиме
<i>TotalKernelTime</i>	Процессорное время, израсходованное процессами задания в режиме ядра
<i>ThisPeriodTotalUserTime</i>	То же, что <i>TotalUserTime</i> , но обнуляется, когда базовые ограничения изменяются вызовом <i>SetInformationJobObject</i> , а флаг <i>JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME</i> не используется
<i>ThisPeriodTotalKernelTime</i>	То же, что <i>ThisPeriodTotalUserTime</i> , но относится к процессорному времени, израсходованному в режиме ядра
<i>TotalPageFaultCount</i>	Общее количество ошибок страниц, вызванных процессами задания
<i>TotalProcesses</i>	Общее число процессов, когда-либо выполнявшихся в этом задании
<i>ActiveProcesses</i>	Текущее количество процессов в задании
<i>TotalTerminatedProcesses</i>	Количество процессов, завершенных из-за превышения ими отведенного лимита процессорного времени

Таблица 5-3. Элементы структуры *JOBOBJECT_BASIC_ACCOUNTING_INFORMATION*

Вы можете извлечь те же сведения вместе с учетной информацией по вводу-выводу, передав *JobObjectBasicAndIoAccountingInformation* во втором параметре и адрес структуры *JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION*:

```
typedef struct _JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION {
    JOBOBJECT_BASIC_ACCOUNTING_INFORMATION BasicInfo;
    IO_COUNTERS IoInfo;
} JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION;
```

Как видите, она просто возвращает *JOBOBJECT_BASIC_ACCOUNTING_INFORMATION* и *IO_COUNTERS*. Последняя структура показана на следующей странице.

```
typedef struct _IO_COUNTERS {
    ULONGLONG ReadOperationCount;
    ULONGLONG WriteOperationCount;
    ULONGLONG OtherOperationCount;
    ULONGLONG ReadTransferCount;
    ULONGLONG WriteTransferCount;
    ULONGLONG OtherTransferCount;
} IO_COUNTERS;
```

Она сообщает о числе операций чтения, записи и перемещения (а также о количестве байтов, переданных при выполнении этих операций). Данные относятся ко всем процессам в задании. Кстати, новая функция *GetProcessIoCounters* позволяет получить ту же информацию о процессах, не входящих ни в какие задания.

```
BOOL GetProcessIoCounters(
    HANDLE hProcess,
    PIO_COUNTERS pIoCounters);
```

QueryInformationJobObject также возвращает набор идентификаторов текущих процессов в задании. Но перед этим Вы должны прикинуть, сколько их там может быть, и выделить соответствующий блок памяти, где поместятся массив идентификаторов и структура *JOBOBJECT_BASIC_PROCESS_ID_LIST*:

```
typedef struct _JOBOBJECT_BASIC_PROCESS_ID_LIST {
    DWORD NumberOfAssignedProcesses;
    DWORD NumberOfProcessIdsInList;
    DWORD ProcessIdList[1];
} JOBOBJECT_BASIC_PROCESS_ID_LIST, *PJOBOBJECT_BASIC_PROCESS_ID_LIST;
```

В итоге, чтобы получить набор идентификаторов текущих процессов в задании, нужно написать примерно такой код:

```
void EnumProcessIdsInJob(HANDLE hjob) {

    // я исхожу из того, что количество процессов
    // в этом задании никогда не превысит 10
    #define MAX_PROCESS_IDS      10

    // определяем размер блока памяти (в байтах)
    // для хранения идентификаторов и структуры
    DWORD cb = sizeof(JOBOBJECT_BASIC_PROCESS_ID_LIST) +
        (MAX_PROCESS_IDS - 1) * sizeof(DWORD);

    // выделяем этот блок памяти
    PJOBOBJECT_BASIC_PROCESS_ID_LIST pjobpil = _alloca(cb);

    // сообщаем функции, на какое максимальное число процессов
    // рассчитана выделенная нами память
    pjobpil->NumberOfAssignedProcesses = MAX_PROCESS_IDS;

    // запрашиваем текущий список идентификаторов процессов
    QueryInformationJobObject(hjob, JobObjectBasicProcessIdList,
        pjobpil, cb, &cb);

    // перечисляем идентификаторы процессов
    for (int x = 0; x < pjobpil->NumberOfProcessIdsInList; x++) {
```

```

    // используем pjobpil->ProcessIdList[x]...
}

// так как для выделения памяти мы вызывали _alloca,
// освобождать память нам не потребуется
}

```

Вот и все, что Вам удастся получить через эти функции, хотя на самом деле операционная система знает о заданиях гораздо больше. Эту информацию, которая хранится в специальных счетчиках, можно извлечь с помощью функций из библиотеки Performance Data Helper (PDH.dll) или через модуль Performance Monitor, подключаемый к Microsoft Management Console (MMC). Рис. 5-3 иллюстрирует некоторые из доступных в системе счетчиков заданий (job object counters), а рис. 5-4 — счетчики, относящиеся к отдельным параметрам заданий (job object details counters). Заметьте, что в задании Jeff содержится четыре процесса: calc, cmd, notepad и wordpad.

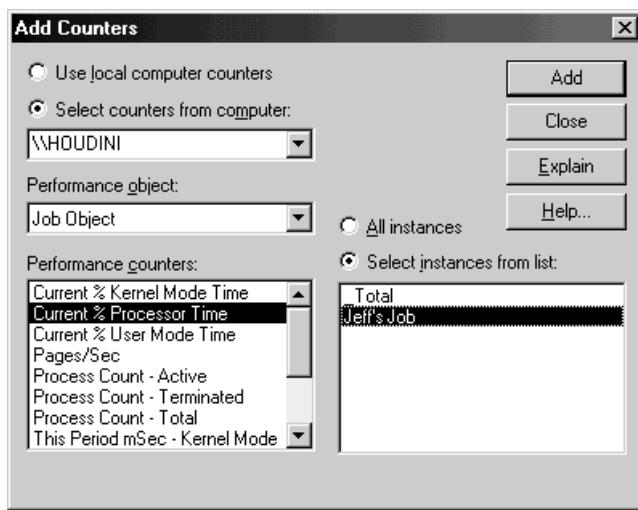


Рис. 5-3. MMC Performance Monitor: счетчики задания

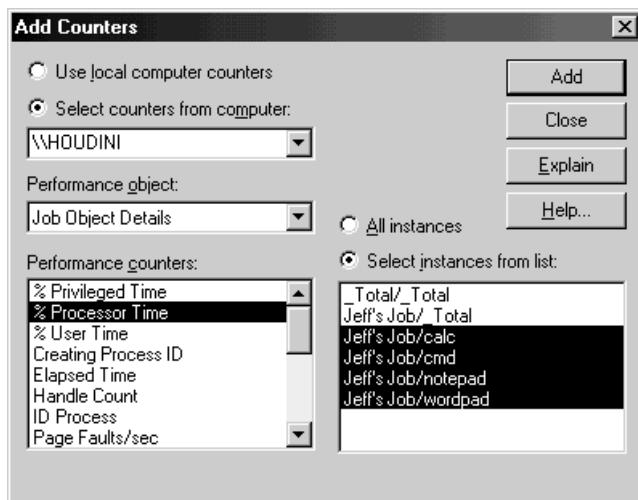


Рис. 5-4. MMC Performance Monitor: счетчики, относящиеся к отдельным параметрам задания

Извлечь сведения из этих счетчиков Вы сможете только для тех заданий, которым были присвоены имена при вызове *CreateJobObject*. По этой причине, наверное, лучше всегда именовать задания, даже если Вы и не собираетесь ссылаться на них по именам из других процессов.

Уведомления заданий

Итак, базовые сведения об объектах-заданиях я изложил. Единственное, что осталось рассмотреть, — уведомления. Допустим, Вам нужно знать, когда завершаются все процессы в задании или заканчивается все отпущенное им процессорное время. Либо выяснить, когда в задании порождается или уничтожается очередной процесс. Если такие уведомления Вас не интересуют (а во многих приложениях они и не нужны), работать с заданиями будет очень легко — не сложнее, чем я уже рассказывал. Но если они все же понадобятся, Вам придется копнуть чуть поглубже.

Информацию о том, все ли выделенное процессорное время исчерпано, получить нетрудно. Объекты-задания не переходят в свободное состояние до тех пор, пока их процессы не израсходуют отведенное процессорное время. Как только оно заканчивается, система уничтожает все процессы в задании и переводит его объект в свободное состояние (*signaled state*). Это событие легко перехватить с помощью *WaitForSingleObject* (или похожей функции). Кстати, потом Вы можете вернуть объект-задание в состояние «занято» (*nonsignaled state*), вызвав *SetInformationJobObject* и выделив ему дополнительное процессорное время.

Когда я только начинал разбираться с заданиями, мне казалось, что объект-задание должен переходить в свободное состояние после завершения всех его процессов. В конце концов, прекращая свою работу, объекты процессов и потоков освобождаются; то же самое вроде бы должно происходить и с заданиями. Но Microsoft предпочла сделать по-другому: объект-задание переходит в свободное состояние после того, как исчерпает выделенное ему время. Поскольку большинство заданий начинает свою работу с одним процессом, который существует, пока не завершатся все его дочерние процессы, Вам нужно просто следить за описателем родительского процесса — он освободится, как только завершится все задание. Моя функция *StartRestrictedProcess* как раз и демонстрирует данный прием.

Но это были лишь простейшие уведомления — более «продвинутые», например о создании или разрушении процесса, получать гораздо сложнее. В частности, Вам придется создать объект ядра «порт завершения ввода-вывода» и связать с ним объект или объекты «задание». После этого нужно будет перевести один или больше потоков в режим ожидания порта завершения.

Создав порт завершения ввода-вывода, Вы сопоставляете с ним задание, вызывая *SetInformationJobObject* следующим образом:

```
JOBOBJECT_ASSOCIATE_COMPLETION_PORT joacp;
joacp.CompletionKey = 1; // любое значение, уникально идентифицирующее
                        // это задание
joacp.CompletionPort = hIOCP; // описатель порта завершения, принимающего
                            // уведомления
SetInformationJobObject(hJob, JobObjectAssociateCompletionPortInformation,
    &joacp, sizeof(joacp));
```

После выполнения этого кода система начнет отслеживать задание и при возникновении событий передавать их порту завершения. (Кстати, Вы можете вызывать *QueryInformationJobObject* и получать ключ завершения и описатель порта, но вряд ли

это Вам когда-нибудь понадобится.) Потоки следят за портом завершения ввода-вывода, вызывая *GetQueuedCompletionStatus*:

```
BOOL GetQueuedCompletionStatus(
    HANDLE hIOCP,
    PDWORD pNumBytesTransferred,
    PULONG_PTR pCompletionKey,
    POVERLAPPED *pOverlapped,
    DWORD dwMilliseconds);
```

Когда эта функция возвращает уведомление о событии задания, **pCompletionKey* содержит значение ключа завершения, заданное при вызове *SetInformationJobObject* для связывания задания с портом завершения. По нему Вы узнаете, в каком из заданий возникло событие. Значение в **pNumBytesTransferred* указывает, какое именно событие произошло (таблица 5-4). В зависимости от конкретного события в **pOverlapped* может возвращаться идентификатор процесса.

Событие	Описание
JOB_OBJECT_MSG_ACTIVE_PROCESS_ZERO	В задании нет работающих процессов
JOB_OBJECT_MSG_END_OF_PROCESS_TIME	Процессорное время, выделенное процессу, исчерпано; процесс завершается, и сообщается его идентификатор
JOB_OBJECT_MSG_ACTIVE_PROCESS_LIMIT	Была попытка превысить ограничение на число активных процессов в задании
JOB_OBJECT_MSG_PROCESS_MEMORY_LIMIT	Была попытка превысить ограничение на объем памяти, которая может быть передана процессу; сообщается идентификатор процесса
JOB_OBJECT_MSG_JOB_MEMORY_LIMIT	Была попытка превысить ограничение на объем памяти, которая может быть передана заданию; сообщается идентификатор процесса
JOB_OBJECT_MSG_NEW_PROCESS	В задание добавлен процесс; сообщается идентификатор процесса
JOB_OBJECT_MSG_EXIT_PROCESS	Процесс завершен; сообщается идентификатор процесса
JOB_OBJECT_MSG_ABNORMAL_EXIT_PROCESS	Процесс завершен из-за необработанного им исключения; сообщается идентификатор процесса
JOB_OBJECT_MSG_END_OF_JOB_TIME	Процессорное время, выделенное заданию, исчерпано; процессы не завершаются, и Вы можете либо возобновить их работу, задав новый лимит по времени, либо самостоятельно завершить процессы, вызвав <i>TerminateJobObject</i>

Таблица 5-4. Уведомления о событиях задания, посылаемые системой связанным с этим заданием порту завершения

И последнее замечание: по умолчанию объект-задание настраивается системой на автоматическое завершение всех его процессов по истечении выделенного ему процессорного времени, а уведомление *JOB_OBJECT_MSG_END_OF_JOB_TIME* не посылается. Если Вы хотите, чтобы объект-задание не уничтожал свои процессы, а просто сообщал о превышении лимита на процессорное время, Вам придется написать примерно такой код:

```
// создаем структуру JOB_OBJECT_END_OF_JOB_TIME_INFORMATION
// и инициализируем ее единственный элемент
```

см. след. стр.

```
JOBOBJECT_END_OF_JOB_TIME_INFORMATION joeobji;  
joeobji.EndOfJobTimeAction = JOB_OBJECT_POST_AT_END_OF_JOB;
```

```
// сообщаем заданию, что ему нужно делать по истечении его времени  
SetInformationJobObject(hJob, JobObjectEndOfJobTimeInformation,  
    &joeobji, sizeof(joeobji));
```

Вы можете указать и другое значение, `JOB_OBJECT_TERMINATE_AT_END_OF_JOB`, но оно задается по умолчанию, еще при создании задания.

Программа-пример JobLab

Эта программа, «05 JobLab.exe» (см. листинг на рис. 5-6), позволяет легко экспериментировать с заданиями. Ее файлы исходного кода и ресурсов находятся в каталоге 05-JobLab на компакт-диске, прилагаемом к книге. После запуска JobLab открывается окно, показанное на рис. 5-5.

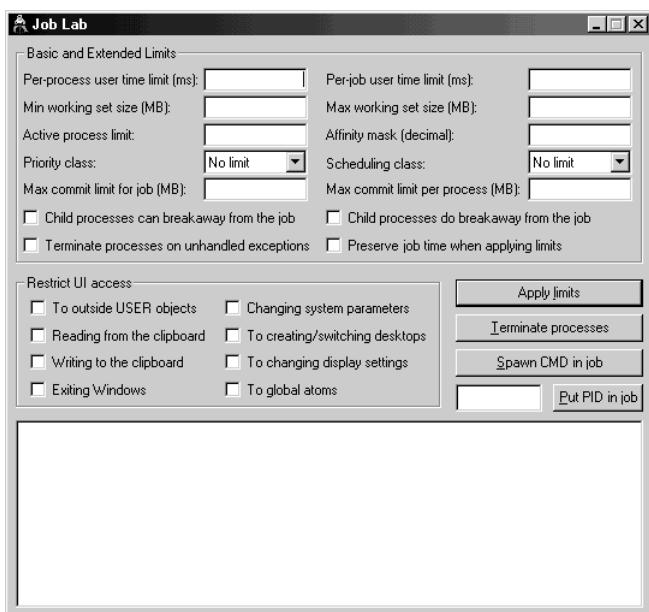


Рис. 5-5. Программа-пример JobLab

Когда процесс инициализируется, он создает объект «задание». Я присваиваю ему имя `JobLab`, чтобы Вы могли наблюдать за ним с помощью MMC Performance Monitor. Моя программа также создает порт завершения ввода-вывода и связывает с ним объект-задание. Это позволяет отслеживать уведомления от задания и отображать их в списке в нижней части окна.

Изначально в задании нет процессов, и никаких ограничений для него не установлено. Поля в верхней части окна позволяют задавать базовые и расширенные ограничения. Все, что от Вас требуется, — ввести в них допустимые значения и щелкнуть кнопку `Apply Limits`. Если Вы оставляете поле пустым, соответствующие ограничения не вводятся. Кроме базовых и расширенных, Вы можете задавать ограничения по пользовательскому интерфейсу. Обратите внимание: помечая флажок `Preserve Job Time When Applying Limits`, Вы не устанавливаете ограничение, а просто получаете возможность изменять ограничения, не сбрасывая значения элементов `ThisPeriod-`

TotalUserTime и *ThisPeriodTotalKernelTime* при запросе базовой учетной информации. Этот флагок становится недоступен при наложении ограничений на процессорное время для отдельных заданий.

Остальные кнопки позволяют управлять заданием по-другому. Кнопка Terminate Processes уничтожает все процессы в задании. Кнопка Spawn CMD In Job запускает командный процессор, сопоставляемый с заданием. Из этого процесса можно запускать дочерние процессы и наблюдать, как они ведут себя, став частью задания. И последняя кнопка, Put PID In Job, позволяет связать существующий свободный процесс с заданием (т. е. включить его в задание).

Список в нижней части окна отображает обновляемую каждые 10 секунд информацию о статусе задания: базовые и расширенные сведения, статистику ввода-вывода, а также пиковые объемы памяти, занимаемые процессом и заданием.

Кроме этой информации, в списке показываются уведомления, поступающие от задания в порт завершения ввода-вывода. (Кстати, вся информация обновляется и при приеме уведомления.)

И еще одно: если Вы измените исходный код и будете создавать безымянный объект ядра «задание», то сможете запускать несколько копий этой программы, создавая тем самым два и более объектов-заданий на одной машине. Это расширит Ваши возможности в экспериментах с заданиями.

Что касается исходного кода, то специально обсуждать его нет смысла — в нем и так достаточно комментариев. Замечу лишь, что в файле Job.h я определил C++-класс CJob, инкапсулирующий объект «задание» операционной системы. Это избавило меня от необходимости передавать туда-сюда описатель задания и позволило уменьшить число операций приведения типов, которые обычно приходится выполнять при вызове функций *QueryInformationJobObject* и *SetInformationJobObject*.



JobLab.cpp

```

/*****
Modуль: JobLab.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"
#include <windowsx.h>
#include <process.h>    // для доступа к _beginthreadex
#include <tchar.h>
#include <stdio.h>
#include "Resource.h"
#include "Job.h"

///////////////////////////////



CJob g_job;           // задание
HWND g_hwnd;          // описатель диалогового окна (доступен всем потокам)
HANDLE g_hIOCP;        // порт завершения, принимающий уведомления от задания
HANDLE g_hThreadIOCP; // поток порта завершения

```

Рис. 5-6. Программа-пример JobLab

см. след. стр.

Рис. 5-6. продолжение

```
// ключи завершения, относящиеся к порту завершения
#define COMPKEY_TERMINATE ((UINT_PTR) 0)
#define COMPKEY_STATUS   ((UINT_PTR) 1)
#define COMPKEY_JOBOBJECT ((UINT_PTR) 2)

///////////////////////////////



DWORD WINAPI JobNotify(PVOID) {
    TCHAR sz[2000];
    BOOL fDone = FALSE;

    while (!fDone) {
        DWORD dwBytesXferred;
        ULONG_PTR CompKey;
        LPOVERLAPPED po;
        GetQueuedCompletionStatus(g_hIOCP,
            &dwBytesXferred, &CompKey, &po, INFINITE);

        // приложение закрывается, выходим из этого потока
        fDone = (CompKey == COMPKEY_TERMINATE);

        HWND hwndLB = FindWindow(NULL, TEXT("Job Lab"));
        hwndLB = GetDlgItem(hwndLB, IDC_STATUS);

        if (CompKey == COMPKEY_JOBOBJECT) {
            lstrcpy(sz, TEXT("--> Notification: "));
            LPTSTR psz = sz + lstrlen(sz);
            switch (dwBytesXferred) {
                case JOB_OBJECT_MSG_END_OF_JOB_TIME:
                    wsprintf(psz, TEXT("Job time limit reached"));
                    break;

                case JOB_OBJECT_MSG_END_OF_PROCESS_TIME:
                    wsprintf(psz, TEXT("Job process (Id=%d) time limit reached"), po);
                    break;

                case JOB_OBJECT_MSG_ACTIVE_PROCESS_LIMIT:
                    wsprintf(psz, TEXT("Too many active processes in job"));
                    break;

                case JOB_OBJECT_MSG_ACTIVE_PROCESS_ZERO:
                    wsprintf(psz, TEXT("Job contains no active processes"));
                    break;

                case JOB_OBJECT_MSG_NEW_PROCESS:
                    wsprintf(psz, TEXT("New process (Id=%d) in job"), po);
                    break;

                case JOB_OBJECT_MSG_EXIT_PROCESS:
                    wsprintf(psz, TEXT("Process (Id=%d) terminated"), po);
                    break;
            }
        }
    }
}
```

Рис. 5-6. продолжение

```

case JOB_OBJECT_MSG_ABNORMAL_EXIT_PROCESS:
    wsprintf(psz, TEXT("Process (Id=%d) terminated abnormally"), po);
    break;

case JOB_OBJECT_MSG_PROCESS_MEMORY_LIMIT:
    wsprintf(psz, TEXT("Process (Id=%d) exceeded memory limit"), po);
    break;

case JOB_OBJECT_MSG_JOB_MEMORY_LIMIT:
    wsprintf(psz,
        TEXT("Process (Id=%d) exceeded job memory limit"), po);
    break;

default:
    wsprintf(psz, TEXT("Unknown notification: %d"), dwBytesXferred);
    break;
}
ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));
CompKey = 1; // принудительное обновление списка
// при получении уведомления

if (CompKey == COMPKEY_STATUS) {
    static int s_nStatusCount = 0;
    _stprintf(sz, TEXT("--> Status Update (%u)'), s_nStatusCount++);
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

    // выводим базовую учетную информацию
    JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION jobai;
    g_job.QueryBasicAccountingInfo(&jobai);

    _stprintf(sz, TEXT("Total Time: User=%I64u, Kernel=%I64u      "))
        TEXT("Period Time: User=%I64u, Kernel=%I64u"),
        jobai.BasicInfo.TotalUserTime.QuadPart,
        jobai.BasicInfo.TotalKernelTime.QuadPart,
        jobai.BasicInfo.ThisPeriodTotalUserTime.QuadPart,
        jobai.BasicInfo.ThisPeriodTotalKernelTime.QuadPart);
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

    _stprintf(sz, TEXT("Page Faults=%u, Total Processes=%u, "))
        TEXT("Active Processes=%u, Terminated Processes=%u"),
        jobai.BasicInfo.TotalPageFaultCount,
        jobai.BasicInfo.TotalProcesses,
        jobai.BasicInfo.ActiveProcesses,
        jobai.BasicInfo.TotalTerminatedProcesses);
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

    // показываем учетную информацию по вводу-выводу
    _stprintf(sz, TEXT("Reads=%I64u (%I64u bytes), "))
        TEXT("Write=%I64u (%I64u bytes), Other=%I64u (%I64u bytes"),
        jobai.IoInfo.ReadOperationCount, jobai.IoInfo.ReadTransferCount,
        jobai.IoInfo.WriteOperationCount, jobai.IoInfo.WriteTransferCount,

```

см. след. стр.

Рис. 5-6. продолжение

```
        jobai.IoInfo.OtherOperationCount, jobai.IoInfo.OtherTransferCount);
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

    // сообщаем пиковые значения объема памяти, использованные
    // процессами и заданиями
    JOBOBJECT_EXTENDED_LIMIT_INFORMATION joeli;
    g_job.QueryExtendedLimitInfo(&joeli);
    _stprintf(sz, TEXT("Peak memory used: Process=%I64u, Job=%I64u"),
              (_int64) joeli.PeakProcessMemoryUsed,
              (_int64) joeli.PeakJobMemoryUsed);
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

    // выводим список идентификаторов процессов
    DWORD dwNumProcesses = 50, dwProcessIdList[50];
    g_job.QueryBasicProcessIdList(dwNumProcesses,
                                  dwProcessIdList, &dwNumProcesses);
    _stprintf(sz, TEXT("PIDs: %s"),
              (dwNumProcesses == 0) ? TEXT("(none)") : TEXT(""));
    for (DWORD x = 0; x < dwNumProcesses; x++) {
        _stprintf(_tcschr(sz, 0), TEXT("%d "), dwProcessIdList[x]);
    }
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));
}

return(0);
}

//////////
```

```
BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_JOBLAB);

    // сохраняем описатель нашего окна, чтобы поток
    // порта завершения мог к нему обращаться
    g_hwnd = hwnd;

    HWND hwndPriorityClass = GetDlgItem(hwnd, IDC_PRIORITYCLASS);
    ComboBox_AddString(hwndPriorityClass, TEXT("No limit"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Idle"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Below normal"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Normal"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Above normal"));
    ComboBox_AddString(hwndPriorityClass, TEXT("High"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Realtime"));
    ComboBox_SetCurSel(hwndPriorityClass, 0); // по умолчанию - "No limit"

    HWND hwndSchedulingClass = GetDlgItem(hwnd, IDC_SCHEDULINGCLASS);
    ComboBox_AddString(hwndSchedulingClass, TEXT("No limit"));
    for (int n = 0; n <= 9; n++) {
        TCHAR szSchedulingClass[2] = { (TCHAR) (TEXT('0') + n), 0 };
        ComboBox_SetString(hwndSchedulingClass, szSchedulingClass);
    }
}
```

Рис. 5-6. продолжение

```

    ComboBox_AddString(hwndSchedulingClass, szSchedulingClass);
}
ComboBox_SetCurSel(hwndSchedulingClass, 0); // по умолчанию - "No limit"
SetTimer(hwnd, 1, 10000, NULL); // обновление каждые 10 секунд
return(TRUE);
}

///////////////////////////////



void Dlg_ApplyLimits(HWND hwnd) {
const int nNanosecondsPerSecond = 100000000;
const int nMillisecondsPerSecond = 1000;
const int nNanosecondsPerMillisecond =
    nNanosecondsPerSecond / nMillisecondsPerSecond;
BOOL f;
__int64 q;
SIZE_T s;
DWORD d;

// устанавливаем базовые и расширенные ограничения
JOBOBJECT_EXTENDED_LIMIT_INFORMATION joeli = { 0 };
joeli.BasicLimitInformation.LimitFlags = 0;

q = GetDlgItemInt(hwnd, IDC_PERPROCESSUSERTIMELIMIT, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_PROCESS_TIME;
    joeli.BasicLimitInformation.PerProcessUserTimeLimit.QuadPart =
        q * nNanosecondsPerMillisecond / 100;
}

q = GetDlgItemInt(hwnd, IDC_PERJOBUSERTIMELIMIT, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_JOB_TIME;
    joeli.BasicLimitInformation.PerJobUserTimeLimit.QuadPart =
        q * nNanosecondsPerMillisecond / 100;
}

s = GetDlgItemInt(hwnd, IDC_MINWORKINGSETSIZE, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_WORKINGSET;
    joeli.BasicLimitInformation.MinimumWorkingSetSize = s * 1024 * 1024;
    s = GetDlgItemInt(hwnd, IDC_MAXWORKINGSETSIZE, &f, FALSE);
    if (f) {
        joeli.BasicLimitInformation.MaximumWorkingSetSize = s * 1024 * 1024;
    } else {
        joeli.BasicLimitInformation.LimitFlags &= ~JOB_OBJECT_LIMIT_WORKINGSET;
        chMB("Both minimum and maximum working set sizes must be set.\n"
             "The working set limits will NOT be in effect.");
    }
}
}

```

см. след. стр.

Рис. 5-6. продолжение

```
d = GetDlgItemInt(hwnd, IDC_ACTIVEPROCESSLIMIT, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_ACTIVE_PROCESS;
    joeli.BasicLimitInformation.ActiveProcessLimit = d;
}

s = GetDlgItemInt(hwnd, IDC_AFFINITYMASK, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_AFFINITY;
    joeli.BasicLimitInformation.Affinity = s;
}

joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_PRIORITY_CLASS;
switch (ComboBox_GetCurSel(GetDlgItem(hwnd, IDC_PRIORITYCLASS))) {
    case 0:
        joeli.BasicLimitInformation.LimitFlags &=
            ~JOB_OBJECT_LIMIT_PRIORITY_CLASS;
        break;

    case 1:
        joeli.BasicLimitInformation.PriorityClass =
            IDLE_PRIORITY_CLASS;
        break;

    case 2:
        joeli.BasicLimitInformation.PriorityClass =
            BELOW_NORMAL_PRIORITY_CLASS;
        break;

    case 3:
        joeli.BasicLimitInformation.PriorityClass =
            NORMAL_PRIORITY_CLASS;
        break;

    case 4:
        joeli.BasicLimitInformation.PriorityClass =
            ABOVE_NORMAL_PRIORITY_CLASS;
        break;

    case 5:
        joeli.BasicLimitInformation.PriorityClass =
            HIGH_PRIORITY_CLASS;
        break;

    case 6:
        joeli.BasicLimitInformation.PriorityClass =
            REALTIME_PRIORITY_CLASS;
        break;
}
```

Рис. 5-6. продолжение

```

int nSchedulingClass =
    ComboBox_GetCurSel(GetDlgItem(hwnd, IDC_SCHEDULINGCLASS));
if (nSchedulingClass > 0) {
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_SCHEDULING_CLASS;
    joeli.BasicLimitInformation.SchedulingClass = nSchedulingClass - 1;
}

s = GetDlgItemInt(hwnd, IDC_MAXCOMMITPERJOB, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_JOB_MEMORY;
    joeli.JobMemoryLimit = s * 1024 * 1024;
}

s = GetDlgItemInt(hwnd, IDC_MAXCOMMITPERPROCESS, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_PROCESS_MEMORY;
    joeli.ProcessMemoryLimit = s * 1024 * 1024;
}

if (IsDlgButtonChecked(hwnd, IDC_CHILDPROCESSESCANBREAKAWAYFROMJOB))
    joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_BREAKAWAY_OK;

if (IsDlgButtonChecked(hwnd, IDC_CHILDPROCESSESDOBREAKAWAYFROMJOB))
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK;

if (IsDlgButtonChecked(hwnd, IDC_TERMINATEPROCESSONEXCEPTIONS))
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION;

f = g_job.SetExtendedLimitInfo(&joeli,
    ((joeli.BasicLimitInformation.LimitFlags & JOB_OBJECT_LIMIT_JOB_TIME)
     != 0) ? FALSE :
    IsDlgButtonChecked(hwnd, IDC_PRESERVEJOBTIMEWHENAPPLYINGLIMITS));
chASSERT(f);

// устанавливаем ограничения, связанные с пользовательским интерфейсом
DWORD jobuir = JOB_OBJECT_UILIMIT_NONE; // "замысловатый" нуль (0)
if (IsDlgButtonChecked(hwnd, IDC_RESTRICTACCESSTOUTSIDEUSEROBJECTS))
    jobuir |= JOB_OBJECT_UILIMIT_HANDLES;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTREADINGCLIPBOARD))
    jobuir |= JOB_OBJECT_UILIMIT_READCLIPBOARD;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTWRITINGCLIPBOARD))
    jobuir |= JOB_OBJECT_UILIMIT_WRITECLIPBOARD;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTEXITWINDOW))
    jobuir |= JOB_OBJECT_UILIMIT_EXITWINDOWS;

```

см. след. стр.

Рис. 5-6. продолжение

```
if (IsDlgButtonChecked(hwnd, IDC_RESTRICTCHANGINGSYSTEMPARAMETERS))
    jobuir |= JOB_OBJECT_UILIMIT_SYSTEMPARAMETERS;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTDESKTOPS))
    jobuir |= JOB_OBJECT_UILIMIT_DESKTOP;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTDISPLAYSETTINGS))
    jobuir |= JOB_OBJECT_UILIMIT_DISPLAYSETTINGS;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTGLOBALATOMS))
    jobuir |= JOB_OBJECT_UILIMIT_GLOBALATOMS;

chVERIFY(g_job.SetBasicUIRestrictions(jobuir));
}

///////////////////////////////
void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    switch (id) {
        case IDCANCEL:
            // пользователь закрывает наше приложение – закрываем и задание
            KillTimer(hwnd, 1);
            g_job.Terminate(0);
            EndDialog(hwnd, id);
            break;

        case IDC_PERJOBUSERTIMELIMIT:
            {
                // устанавливая новое ограничение по времени,
                // нужно сбросить ранее указанное время для задания
                BOOL f;
                GetDlgItemInt(hwnd, IDC_PERJOBUSERTIMELIMIT, &f, FALSE);
                EnableWindow(
                    GetDlgItem(hwnd, IDC_PRESERVEJOBTIMEWHENAPPLYINGLIMITS), !f);
            }
            break;

        case IDC_APPLYLIMITS:
            Dlg_ApplyLimits(hwnd);
            PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
            break;

        case IDC_TERMINATE:
            g_job.Terminate(0);
            PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
            break;

        case IDC_SPAWNCMDINJOB:
            {
                // порождаем процесс командного процессора и помещаем его в задание
```

Рис. 5-6. продолжение

```

STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
TCHAR sz[] = TEXT("CMD");
CreateProcess(NULL, sz, NULL, NULL,
    FALSE, CREATE_SUSPENDED, NULL, NULL, &si, &pi);
g_job.AssignProcess(pi.hProcess);
ResumeThread(pi.hThread);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
}
PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
break;

case IDC_ASSIGNPROCESSTOJOB:
{
    DWORD dwProcessId = GetDlgItemInt(hwnd, IDC_PROCESSID, NULL, FALSE);
    HANDLE hProcess = OpenProcess(
        PROCESS_SET_QUOTA | PROCESS_TERMINATE, FALSE, dwProcessId);
    if (hProcess != NULL) {
        chVERIFY(g_job.AssignProcess(hProcess));
        CloseHandle(hProcess);
    } else chMB("Could not assign process to job.");
}
PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
break;
}

///////////////////////////////
void WINAPI Dlg_OnTimer(HWND hwnd, UINT id) {
    PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
}

/////////////////////////////
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
    switch (uMsg) {
        chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
        chHANDLE_DLGMMSG(hwnd, WM_TIMER, Dlg_OnTimer);
        chHANDLE_DLGMMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
    }
    return(FALSE);
}

/////////////////////////////

```

см. след. стр.

Рис. 5-6. продолжение

```
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {  
  
    // создаем порт завершения, который будет принимать уведомления от задания  
    g_hIOCP = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);  
  
    // создаем поток, ждущий порт завершения  
    g_hThreadIOCP = chBEGINTHREADEX(NULL, 0, JobNotify, NULL, 0, NULL);  
  
    // создаем объект-задание  
    g_job.Create(NULL, TEXT("JobLab"));  
    g_job.SetEndOfJobInfo(JOB_OBJECT_POST_AT_END_OF_JOB);  
    g_job.AssociateCompletionPort(g_hIOCP, COMPKEY_JOBOBJECT);  
  
    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_JOBLAB), NULL, Dlg_Proc);  
  
    // передаем специальный ключ, заставляющий поток  
    // порта завершения закончить работу  
    PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_TERMINATE, NULL);  
  
    // ждем, когда завершится его поток  
    WaitForSingleObject(g_hThreadIOCP, INFINITE);  
  
    // проводим должную очистку  
    CloseHandle(g_hIOCP);  
    CloseHandle(g_hThreadIOCP);  
  
    // ПРИМЕЧАНИЕ: задание закрывается вызовом деструктора g_job  
    return(0);  
}  
  
//////////////////////////// Конец файла //////////////////////////////
```

Job.h

```
*****  
Модуль: Job.h  
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)  
*****  
  
#pragma once  
  
//////////////////////////////  
  
#include <malloc.h>      // для доступа к _alloca  
  
//////////////////////////////  
  
class CJob {  
public:  
    CJob(HANDLE hJob = NULL);  
    ~CJob();
```

Рис. 5-6. продолжение

```

operator HANDLE() const { return(m_hJob); }

// функции, создающие или открывающие объект-задание
BOOL Create(LPSECURITY_ATTRIBUTES psa = NULL, LPCTSTR pszName = NULL);
BOOL Open(LPCTSTR pszName, DWORD dwDesiredAccess,
          BOOL fInheritHandle = FALSE);

// функции, манипулирующие объектом-заданием
BOOL AssignProcess(HANDLE hProcess);
BOOL Terminate(UINT uExitCode = 0);

// функции, налагающие ограничения на задания
BOOL SetExtendedLimitInfo(PJOBOBJECT_EXTENDED_LIMIT_INFORMATION pjoeli,
                           BOOL fPreserveJobTime = FALSE);
BOOL SetBasicUIRestrictions(DWORD fdwLimits);
BOOL GrantUserHandleAccess(HANDLE hUserObj, BOOL fGrant = TRUE);
BOOL SetSecurityLimitInfo(PJOBOBJECT_SECURITY_LIMIT_INFORMATION pjosli);

// функции, запрашивающие сведения об ограничениях
BOOL QueryExtendedLimitInfo(PJOBOBJECT_EXTENDED_LIMIT_INFORMATION pjoeli);
BOOL QueryBasicUIRestrictions(PDWORD pfdwRestrictions);
BOOL QuerySecurityLimitInfo(PJOBOBJECT_SECURITY_LIMIT_INFORMATION pjosli);

// функции, запрашивающие статусную информацию о задании
BOOL QueryBasicAccountingInfo(
    PJOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION pjobai);
BOOL QueryBasicProcessIdList(DWORD dwMaxProcesses,
                             PDWORD pdwProcessIdList, PDWORD pdwProcessesReturned = NULL);

// функции, работающие с уведомлениями задания
BOOL AssociateCompletionPort(HANDLE hIOCP, ULONG_PTR CompKey);
BOOL QueryAssociatedCompletionPort(
    PJOBOBJECT_ASSOCIATE_COMPLETION_PORT pjoacp);
BOOL SetEndOfJobInfo(
    DWORD fdwEndOfJobInfo = JOB_OBJECT_TERMINATE_AT_END_OF_JOB);
BOOL QueryEndOfJobTimeInfo(PDWORD pfdwEndOfJobTimeInfo);

private:
    HANDLE m_hJob;
};

/////////////////////////////// inline CJob::CJob(HANDLE hJob) {
    m_hJob = hJob;
}

///////////////////////////////

```

см. след. стр.

Рис. 5-6. продолжение

```
inline CJob::~CJob() {

    if (m_hJob != NULL)
        CloseHandle(m_hJob);
}

///////////////////////////////



inline BOOL CJob::Create(PSECURITY_ATTRIBUTES psa, PCTSTR pszName) {

    m_hJob = CreateJobObject(psa, pszName);
    return(m_hJob != NULL);
}

///////////////////////////////



inline BOOL CJob::Open(
    PCTSTR pszName, DWORD dwDesiredAccess, BOOL fInheritHandle) {

    m_hJob = OpenJobObject(dwDesiredAccess, fInheritHandle, pszName);
    return(m_hJob != NULL);
}

///////////////////////////////



inline BOOL CJob::AssignProcess(HANDLE hProcess) {

    return(AssignProcessToJobObject(m_hJob, hProcess));
}

///////////////////////////////



inline BOOL CJob::AssociateCompletionPort(HANDLE hIOCP, ULONG_PTR CompKey) {

    JOBOBJECT_ASSOCIATE_COMPLETION_PORT joacp = { (PVOID) CompKey, hIOCP };
    return(SetInformationJobObject(m_hJob,
        JobObjectAssociateCompletionPortInformation, &joacp, sizeof(joacp)));
}

///////////////////////////////



inline BOOL CJob::SetExtendedLimitInfo(
    PJOBOBJECT_EXTENDED_LIMIT_INFORMATION pjoeli, BOOL fPreserveJobTime) {

    if (fPreserveJobTime)
        pjoeli->BasicLimitInformation.LimitFlags |=
            JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME;

    // если Вы хотите сохранить информацию о времени задания,
    // флаг JOB_OBJECT_LIMIT_JOB_TIME нужно убрать
    const DWORD fdwFlagTest =
```

Рис. 5-6. продолжение

```

(JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME | JOB_OBJECT_LIMIT_JOB_TIME);

if ((pjoeli->BasicLimitInformation.LimitFlags & fdwFlagTest)
    == fdwFlagTest) {
    // ошибка, так как указаны два взаимоисключающих флага
    DebugBreak();
}

return(SetInformationJobObject(m_hJob,
    JobObjectExtendedLimitInformation, pjoeli, sizeof(*pjoeli)));
}

///////////////////////////////
inline BOOL CJob::SetBasicUIRestrictions(DWORD fdwLimits) {

    JOBOBJECT_BASIC_UI_RESTRICTIONS jobuir = { fdwLimits };
    return(SetInformationJobObject(m_hJob,
        JobObjectBasicUIRestrictions, &jobuir, sizeof(jobuir)));
}

///////////////////////////////
inline BOOL CJob::SetEndOfJobInfo(DWORD fdwEndOfJobInfo) {

    JOBOBJECT_END_OF_JOB_TIME_INFORMATION joeoji = { fdwEndOfJobInfo };
    joeoji.EndOfJobTimeAction = fdwEndOfJobInfo;
    return(SetInformationJobObject(m_hJob,
        JobObjectEndOfJobTimeInformation, &joeoji, sizeof(joeoji)));
}

///////////////////////////////
inline BOOL CJob::SetSecurityLimitInfo(
    PJOBOBJECT_SECURITY_LIMIT_INFORMATION pjosli) {

    return(SetInformationJobObject(m_hJob,
        JobObjectSecurityLimitInformation, pjosli, sizeof(*pjosl)));
}

///////////////////////////////
inline BOOL CJob::QueryAssociatedCompletionPort(
    PJOBOBJECT_ASSOCIATE_COMPLETION_PORT pjoacp) {

    return(QueryInformationJobObject(m_hJob,
        JobObjectAssociateCompletionPortInformation, pjoacp, sizeof(*pjoacp),
        NULL));
}

/////////////////////////////

```

см. след. стр.

Рис. 5-6. продолжение

```
inline BOOL CJob::QueryBasicAccountingInfo(
    PJOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION pjobai) {

    return(QueryInformationJobObject(m_hJob,
        JobObjectBasicAndIoAccountingInformation, pjobai, sizeof(*pjobai),
        NULL));
}

///////////////////////////////



inline BOOL CJob::QueryExtendedLimitInfo(
    PJOBOBJECT_EXTENDED_LIMIT_INFORMATION pjoeli) {

    return(QueryInformationJobObject(m_hJob, JobObjectExtendedLimitInformation,
        pjoeli, sizeof(*pjoeli), NULL));
}

///////////////////////////////



inline BOOL CJob::QueryBasicProcessIdList(DWORD dwMaxProcesses,
    PDWORD pdwProcessIdList, PDWORD pdwProcessesReturned) {

    // определяем требуемый объем памяти в байтах
    DWORD cb = sizeof(JOBOBJECT_BASIC_PROCESS_ID_LIST) +
        (sizeof(DWORD) * (dwMaxProcesses - 1));

    // выделяем эти байты из стека
    PJOBOBJECT_BASIC_PROCESS_ID_LIST pjobpil =
        (PJOBOBJECT_BASIC_PROCESS_ID_LIST) _alloca(cb);

    // Успешно ли прошла эта операция? Если да, идем дальше.
    BOOL fOk = (pjobpil != NULL);

    if (fOk) {
        pjobpil->NumberofProcessIdsInList = dwMaxProcesses;
        fOk = ::QueryInformationJobObject(m_hJob, JobObjectBasicProcessIdList,
            pjobpil, cb, NULL);

        if (fOk) {
            // у нас появилась информация, возвращаем ее тому, кто ее запрашивал
            if (pdwProcessesReturned != NULL)
                *pdwProcessesReturned = pjobpil->NumberofProcessIdsInList;

            CopyMemory(pdwProcessIdList, pjobpil->ProcessIdList,
                sizeof(DWORD) * pjobpil->NumberofProcessIdsInList);
        }
    }
    return(fOk);
}

///////////////////////////////
```

Рис. 5-6. продолжение

```

inline BOOL CJob::QueryBasicUIRestrictions(PDWORD pfdwRestrictions) {

    JOBOBJECT_BASIC_UI_RESTRICTIONS jobuir;
    BOOL fOk = QueryInformationJobObject(m_hJob, JobObjectBasicUIRestrictions,
        &jobuir, sizeof(jobuir), NULL);
    if (fOk)
        *pfdwRestrictions = jobuir.UIRestrictionsClass;
    return(fOk);
}

/////////////////////////////// Конец файла ///////////////////////////////

```

```

inline BOOL CJob::QueryEndOfJobTimeInfo(PDWORD pfdwEndOfJobTimeInfo) {

    JOBOBJECT_END_OF_JOB_TIME_INFORMATION joeoji;
    BOOL fOk = QueryInformationJobObject(m_hJob, JobObjectBasicUIRestrictions,
        &joeoji, sizeof(joeoji), NULL);
    if (fOk)
        *pfdwEndOfJobTimeInfo = joeoji.EndOfJobTimeAction;
    return(fOk);
}

/////////////////////////////// Конец файла ///////////////////////////////

```

```

inline BOOL CJob::QuerySecurityLimitInfo(
    PJOBOBJECT_SECURITY_LIMIT_INFORMATION pjosli) {

    return(QueryInformationJobObject(m_hJob, JobObjectSecurityLimitInformation,
        pjosli, sizeof(*pjosl), NULL));
}

```

```

/////////////////////////////// Конец файла ///////////////////////////////

```

```

inline BOOL CJob::Terminate(UINT uExitCode) {

    return(TerminateJobObject(m_hJob, uExitCode));
}

```

```

/////////////////////////////// Конец файла ///////////////////////////////

```

```

inline BOOL CJob::GrantUserHandleAccess(HANDLE hUserObj, BOOL fGrant) {

    return(UserHandleGrantAccess(hUserObj, m_hJob, fGrant));
}

```

```

/////////////////////////////// Конец файла ///////////////////////////////

```

Базовые сведения о потоках

Тематика, связанная потоками, очень важна, потому что в любом процессе должен быть хотя бы один поток. В этой главе концепции потоков будут рассмотрены гораздо подробнее. В частности, я объясню, в чем разница между процессами и потоками и для чего они предназначены. Также я расскажу о том, как система использует объекты ядра «поток» для управления потоками. Подобно процессам, потоки обладают определенными свойствами, поэтому мы поговорим о функциях, позволяющих обращаться к этим свойствам и при необходимости модифицировать их. Кроме того, Вы узнаете о функциях, предназначенных для создания (порождения) дополнительных потоков в системе.

В главе 4 я говорил, что процесс фактически состоит из двух компонентов: объекта ядра «процесс» и адресного пространства. Так вот, любой поток тоже состоит из двух компонентов:

- объекта ядра, через который операционная система управляет потоком. Там же хранится статистическая информация о потоке;
- стека потока, который содержит параметры всех функций и локальные переменные, необходимые потоку для выполнения кода. (О том, как система управляет стеком потока, я расскажу в главе 16.)

В той же главе 4 я упомянул, что процессы инертны. Процесс ничего не исполняет, он просто служит контейнером потоков. Потоки всегда создаются в контексте какого-либо процесса, и вся их жизнь проходит только в его границах. На практике это означает, что потоки исполняют код и манипулируют данными в адресном пространстве процесса. Поэтому, если два и более потоков выполняется в контексте одного процесса, все они делят одно адресное пространство. Потоки могут исполнять один и тот же код и манипулировать одними и теми же данными, а также совместно использовать описатели объектов ядра, поскольку таблица описателей создается не в отдельных потоках, а в процессах.

Как видите, процессы используют куда больше системных ресурсов, чем потоки. Причина кроется в адресном пространстве. Создание виртуального адресного пространства для процесса требует значительных системных ресурсов. При этом ведется масса всяческой статистики, на что уходит немало памяти. В адресное пространство загружаются EXE- и DLL-файлы, а значит, нужны файловые ресурсы. С другой стороны, потоку требуются лишь соответствующий объект ядра и стек; объем статистических сведений о потоке невелик и много памяти не занимает.

Так как потоки расходуют существенно меньше ресурсов, чем процессы, старайтесь решать свои задачи за счет использования дополнительных потоков и избегайте создания новых процессов. Только не принимайте этот совет за жесткое правило — многие проекты как раз лучше реализовать на основе множества процессов. Нужно просто помнить об издержках и соразмерять цель и средства.

Прежде чем мы углубимся в скучные, но крайне важные концепции, давайте обсудим, как правильно пользоваться потоками, разрабатывая архитектуру приложения.

В каких случаях потоки создаются

Поток (thread) определяет последовательность исполнения кода в процессе. При инициализации процесса система всегда создает первичный поток. Начинаясь со стартового кода из библиотеки C/C++, который в свою очередь вызывает входную функцию (*WinMain*, *wWinMain*, *main* или *wmain*) из Вашей программы, он живет до того момента, когда входная функция возвращает управление стартовому коду и тот вызывает функцию *ExitProcess*. Большинство приложений обходится единственным, первичным потоком. Однако процессы могут создавать дополнительные потоки, что позволяет им эффективнее выполнять свою работу.

У каждого компьютера есть чрезвычайно мощный ресурс — центральный процессор. И нет абсолютно никаких причин тому, чтобы этот процессор простоявал (не считая экономии электроэнергии). Чтобы процессор всегда был при деле, Вы нагружаете его самыми разнообразными задачами. Вот несколько примеров.

- Вы активизируете службу индексации данных (content indexing service) Windows 2000. Она создает поток с низким приоритетом, который, периодически пробуждаясь, индексирует содержимое файлов на дисковых устройствах Вашего компьютера. Чтобы найти какой-либо файл, Вы открываете окно Search Results (щелкнув кнопку Start и выбрав из меню Search команду For Files Or Folders) и вводите в поле Containing Text нужные критерии поиска. После этого начинается поиск по индексу, и на экране появляется список файлов, удовлетворяющих этим критериям. Служба индексации данных значительно увеличивает скорость поиска, так как при ее использовании больше не требуется открывать, сканировать и закрывать каждый файл на диске.
- Вы запускаете программу для дефрагментации дисков, поставляемую с Windows 2000. Обычно утилиты такого рода предлагают массу настроек для администрирования, в которых средний пользователь совершенно не разбирается, — например, когда и как часто проводить дефрагментацию. Благодаря потокам с более низким приоритетом Вы можете пользоваться этой программой в фоновом режиме и дефрагментировать диски в те моменты, когда других дел у системы нет.
- Нетрудно представить будущую версию компилятора, способную автоматически компилировать файлы исходного кода в паузах, возникающих при наборе текста программы. Тогда предупреждения и сообщения об ошибках появлялись бы практически в режиме реального времени, и Вы тут же видели бы, в чем Вы ошиблись. Самое интересное, что Microsoft Visual Studio в какой-то мере уже умеет это делать, — обратите внимание на секцию ClassView в Workspace.
- Электронные таблицы пересчитывают данные в фоновом режиме.
- Текстовые процессоры разбивают текст на страницы, проверяют его на орографические и грамматические ошибки, а также печатают в фоновом режиме.
- Файлы можно копировать на другие носители тоже в фоновом режиме.
- Web-браузеры способны взаимодействовать с серверами в фоновом режиме. Благодаря этому пользователь может перейти на другой Web-узел, не дожидаясь, когда будут получены результаты с текущего Web-узла.

Одна важная вещь, на которую Вы должны были обратить внимание во всех этих примерах, заключается в том, что поддержка многопоточности позволяет упростить пользовательский интерфейс приложения. Если компилятор ведет сборку Вашей программы в те моменты, когда Вы делаете паузы в наборе ее текста, отпадает необходимость в командах меню Build. То же самое относится к командам Check Spelling и Check Grammar в текстовых процессорах.

В примере с Web-браузером выделение ввода-вывода (сетевого, файлового или какого-то другого) в отдельный поток обеспечивает « отзывчивость » пользовательского интерфейса приложения даже при интенсивной передаче данных. Вообразите приложение, которое сортирует записи в базе данных, печатает документ или копирует файлы. Возложив любую из этих задач, так или иначе связанных с вводом-выводом, на отдельный поток, пользователь может по-прежнему работать с интерфейсом приложения и при необходимости отменить операцию, выполняемую в фоновом режиме.

Многопоточное приложение легче масштабируется. Как Вы увидите в следующей главе, каждый поток можно закрепить за определенным процессором. Так что, если в Вашем компьютере имеется два процессора, а в приложении — два потока, оба процессора будут при деле. И фактически Вы сможете выполнять две задачи одновременно.

В каждом процессе есть хотя бы один поток. Даже не делая ничего особенного в приложении, Вы уже выигрываете только от того, что оно выполняется в многопоточной операционной системе. Например, Вы можете собирать программу и одновременно пользоваться текстовым процессором (довольно часто я так и работаю). Если в компьютере установлено два процессора, то сборка выполняется на одном из них, а документ обрабатывается на другом. Иначе говоря, какого-либо падения производительности не наблюдается. И кроме того, если компилятор из-за той или иной ошибки входит в бесконечный цикл, на остальных процессах это никак не отражается. (Конечно, о программах для MS-DOS и 16-разрядной Windows речь не идет.)

И в каких случаях потоки не создаются

До сих пор я пел одни дифирамбы многопоточным приложениям. Но, несмотря на все преимущества, у них есть и свои недостатки. Некоторые разработчики почему-то считают, будто *любую* проблему можно решить, разбив программу на отдельные потоки. Трудно совершить большую ошибку!

Потоки — вещь невероятно полезная, когда ими пользуются с умом. Увы, решая старые проблемы, можно создать себе новые. Допустим, Вы разрабатываете текстовый процессор и хотите выделить функциональный блок, отвечающий за распечатку, в отдельный поток. Идея вроде неплоха: пользователь, отправив документ на распечатку, может сразу вернуться к редактированию. Но задумайтесь вот над чем: значит, информация в документе может быть изменена *при распечатке* документа? Как видите, теперь перед Вами совершенно новая проблема, с которой прежде сталкиваться не приходилось. Тут-то и подумаешь, а стоит ли выделять печать в отдельный поток, зачем искать лишних приключений? Но давайте разрешим при распечатке редактирование любых документов, кроме того, который печатается в данный момент. Или так: скопируем документ во временный файл и отправим на печать именно его, а пользователь пусть редактирует оригинал в свое удовольствие. Когда распечатка временного файла закончится, мы его удалим — вот и все.

Еще одно узкое место, где неправильное применение потоков может привести к появлению проблем, — разработка пользовательского интерфейса в приложении. В подавляющем большинстве программ все компоненты пользовательского интерфей-

са (окна) обрабатываются одним и тем же потоком. И дочерние окна любого окна определенно должен создавать только один поток. Создание разных окон в разных потоках иногда имеет смысл, но такие случаи действительно редки.

Обычно в приложении существует один поток, отвечающий за поддержку пользовательского интерфейса, — он создает все окна и содержит цикл *GetMessage*. Любые другие потоки в процессе являются рабочими (т. е. отвечают за вычисления, ввод-вывод и другие операции) и не создают никаких окон. Поток пользовательского интерфейса, как правило, имеет более высокий приоритет, чем рабочие потоки, — это нужно для того, чтобы он всегда быстро реагировал на действия пользователя.

Несколько потоков пользовательского интерфейса в одном процессе можно обнаружить в таких приложениях, как Windows Explorer. Он создает отдельный поток для каждого окна папки. Это позволяет копировать файлы из одной папки в другую и попутно просматривать содержимое еще какой-то папки. Кроме того, если какая-то ошибка в Explorer приводит к краху одного из его потоков, прочие потоки остаются работоспособны, и Вы можете пользоваться соответствующими окнами, пока не сделаете что-нибудь такое, из-за чего рухнут и они. (Подробнее о потоках и пользовательском интерфейсе см. главы 26 и 27.)

В общем, мораль этого вступления такова: многопоточность следует использовать разумно. Не создавайте несколько потоков только потому, что это возможно. Многие полезные и мощные программы по-прежнему строятся на основе одного первично-го потока, принадлежащего процессу.

Ваша первая функция потока

Каждый поток начинает выполнение с некоей входной функции. В первичном пото-ке таковой является *main*, *wmain*, *WinMain* или *wWinMain*. Если Вы хотите создать вторичный поток, в нем тоже должна быть входная функция, которая выглядит пример-но так:

```
DWORD WINAPI ThreadFunc(PVOID pvParam) {
    DWORD dwResult = 0;
    :
    return(dwResult);
}
```

Функция потока может выполнять любые задачи. Рано или поздно она закончит свою работу и вернет управление. В этот момент Ваш поток остановится, память, от-веденная под его стек, будет освобождена, а счетчик пользователей его объекта ядра «поток» уменьшится на 1. Когда счетчик обнулится, этот объект ядра будет разрушен. Но, как и объект ядра «процесс», он может жить гораздо дольше, чем сопоставленный с ним поток.

А теперь поговорим о самых важных вещах, касающихся функций потоков.

- В отличие от входной функции первичного потока, у которой должно быть одно из четырех имен: *main*, *wmain*, *WinMain* или *wWinMain*, — функцию пото-ка можно назвать как угодно. Однако, если в программе несколько функций потоков, Вы должны присвоить им разные имена, иначе компилятор или компоновщик решит, что Вы создаете несколько реализаций единственной функции.
- Поскольку входным функциям первичного потока передаются строковые па-раметры, они существуют в ANSI- и Unicode-версиях: *main* — *wmain* и *WinMain* —

wWinMain. Но функциям потоков передается единственный параметр, смысл которого определяется Вами, а не операционной системой. Поэтому здесь нет проблем с ANSI/Unicode.

- Функция потока должна возвращать значение, которое будет использоваться как код завершения потока. Здесь полная аналогия с библиотекой C/C++: код завершения первичного потока становится кодом завершения процесса.
- Функции потоков (да и все Ваши функции) должны по мере возможности обходиться своими параметрами и локальными переменными. Так как к статической или глобальной переменной могут одновременно обратиться несколько потоков, есть риск повредить ее содержимое. Однако параметры и локальные переменные создаются в стеке потока, поэтому они в гораздо меньшей степени подвержены влиянию другого потока.

Вот Вы и узнали, как должна быть реализована функция потока. Теперь рассмотрим, как заставить операционную систему создать поток, который выполнит эту функцию.

Функция *CreateThread*

Мы уже говорили, как при вызове функции *CreateProcess* появляется на свет первичный поток процесса. Если Вы хотите создать дополнительные потоки, нужно вызвать из первичного потока функцию *CreateThread*:

```
HANDLE CreateThread(  
    PSECURITY_ATTRIBUTES psa;  
    DWORD cbStack;  
    PTHREAD_START_ROUTINE pfnStartAddr;  
    PVOID pvParam;  
    DWORD fdwCreate;  
    PDWORD pdwThreadID);
```

При каждом вызове этой функции система создает объект ядра «поток». Это не сам поток, а компактная структура данных, которая используется операционной системой для управления потоком и хранит статистическую информацию о потоке. Так что объект ядра «поток» — полный аналог объекта ядра «процесс».

Система выделяет память под стек потока из адресного пространства процесса. Новый поток выполняется в контексте того же процесса, что и родительский поток. Поэтому он получает доступ ко всем описателям объектов ядра, всей памяти и стекам всех потоков в процессе. За счет этого потоки в рамках одного процесса могут легко взаимодействовать друг с другом.



CreateThread — это Windows-функция, создающая поток. Но никогда не вызывайте ее, если Вы пишете код на C/C++. Вместо нее Вы должны использовать функцию *_beginthreadex* из библиотеки Visual C++. (Если Вы работаете с другим компилятором, он должен поддерживать свой эквивалент функции *CreateThread*.) Что именно делает *_beginthreadex* и почему это так важно, я объясню потом.

О'кэй, общее представление о функции *CreateThread* Вы получили. Давайте рассмотрим все ее параметры.

Параметр *psa*

Параметр *psa* является указателем на структуру SECURITY_ATTRIBUTES. Если Вы хотите, чтобы объекту ядра «поток» были присвоены атрибуты защиты по умолчанию (что чаще всего и бывает), передайте в этом параметре NULL. А чтобы дочерние процессы смогли наследовать описатель этого объекта, определите структуру SECURITY_ATTRIBUTES и инициализируйте ее элемент *bInheritHandle* значением TRUE (см. главу 3).

Параметр *cbStack*

Этот параметр определяет, какую часть адресного пространства поток сможет использовать под свой стек. Каждому потоку выделяется отдельный стек. Функция *CreateProcess*, запуская приложение, вызывает *CreateThread*, и та инициализирует первичный поток процесса. При этом *CreateProcess* заносит в параметр *cbStack* значение, хранящееся в самом исполняемом файле. Управлять этим значением позволяет ключ /STACK компоновщика:

/STACK:[*reserve*] [,*commit*]

Аргумент *reserve* определяет объем адресного пространства, который система должна зарезервировать под стек потока (по умолчанию — 1 Мб). Аргумент *commit* задает объем физической памяти, который изначально передается области, зарезервированной под стек (по умолчанию — 1 страница). По мере исполнения кода в потоке Вам, весьма вероятно, понадобится отвести под стек больше одной страницы памяти. При переполнении стека возникнет исключение. (О стеке потока и исключениях, связанных с его переполнением, см. главу 16, а об общих принципах обработки исключений — главу 23.) Перехватив это исключение, система передаст зарезервированному пространству еще одну страницу (или столько, сколько указано в аргументе *commit*). Такой механизм позволяет динамически увеличивать размер стека лишь по необходимости.

Если Вы, обращаясь к *CreateThread*, передаете в параметре *cbStack* ненулевое значение, функция резервирует всю указанную Вами память. Ее объем определяется либо значением параметра *cbStack*, либо значением, заданным в ключе /STACK компоновщика (выбирается большее из них). Но передается стеку лишь тот объем памяти, который соответствует значению в *cbStack*. Если же Вы передаете в параметре *cbStack* нулевое значение, *CreateThread* создает стек для нового потока, используя информацию, встроенную компоновщиком в EXE-файл.

Значение аргумента *reserve* устанавливает верхний предел для стека, и это ограничение позволяет прекращать деятельность функций с бесконечной рекурсией. Допустим, Вы пишете функцию, которая рекурсивно вызывает сама себя. Предположим также, что в функции есть «жучок», приводящий к бесконечной рекурсии. Всякий раз, когда функция вызывает сама себя, в стеке создается новый стековый фрейм. Если бы система не позволяла ограничивать максимальный размер стека, рекурсивная функция так и вызывала бы сама себя до бесконечности, а стек поглотил бы все адресное пространство процесса. Задавая же определенный предел, Вы, во-первых, предотвращаете разрастание стека до гигантских объемов и, во-вторых, гораздо быстрее узнаете о наличии ошибки в своей программе. (Программа-пример Summation в главе 16 продемонстрирует, как перехватывать и обрабатывать переполнение стека в приложениях.)

Параметры *pfnStartAddr* и *pvParam*

Параметр *pfnStartAddr* определяет адрес функции потока, с которой должен будет начать работу создаваемый поток, а параметр *pvParam* идентичен параметру *pvParam* функции потока. *CreateThread* лишь передает этот параметр по эстафете той функции, с которой начинается выполнение создаваемого потока. Таким образом, данный параметр позволяет передавать функции потока какое-либо инициализирующее значение. Оно может быть или просто числовым значением, или указателем на структуру данных с дополнительной информацией.

Вполне допустимо и даже полезно создавать несколько потоков, у которых в качестве входной точки используется адрес одной и той же функции. Например, можно реализовать Web-сервер, который обрабатывает каждый клиентский запрос в отдельном потоке. При создании каждому потоку передается свое значение *pvParam*.

Учтите, что Windows — операционная система с вытесняющей многозадачностью, а следовательно, новый поток и поток, вызвавший *CreateThread*, могут выполняться одновременно. В связи с этим возможны проблемы. Остерегайтесь, например, такого кода:

```
DWORD WINAPI FirstThread(PVOID pvParam) {
    // инициализируем переменную, которая содержится в стеке
    int x = 0;
    DWORD dwThreadId;

    // создаем новый поток
    HANDLE hThread = CreateThread(NULL, 0, SecondThread, (PVOID) &x,
        0, &dwThreadId);

    // мы больше не ссылаемся на новый поток,
    // поэтому закрываем свой описатель этого потока
    CloseHandle(hThread);

    // Наш поток закончил работу.
    // ОШИБКА: его стек будет разрушен, но SecondThread
    // может попытаться обратиться к нему.
    return(0);
}

DWORD WINAPI SecondThread(PVOID pvParam) {
    // здесь выполняется какая-то длительная обработка
    :
    // Пытаемся обратиться к переменной в стеке FirstThread.
    // ПРИМЕЧАНИЕ: это может привести к ошибке общей защиты –
    // нарушению доступа!
    * ((int *) pvParam) = 5;
    :
    return(0);
}
```

Не исключено, что в приведенном коде *FirstThread* закончит свою работу до того, как *SecondThread* присвоит значение 5 переменной *x* из *FirstThread*. Если так и будет, *SecondThread* не узнает, что *FirstThread* больше не существует, и попытается изменить содержимое какого-то участка памяти с недействительным теперь адресом. Это неизбежно вызовет нарушение доступа: стек первого потока уничтожен по завершении

FirstThread. Что же делать? Можно объявить *x* статической переменной, и компилятор отведет память для хранения переменной *x* не в стеке, а в разделе данных приложения (application's data section). Но тогда функция станет нереентрабельной. Иначе говоря, в этом случае Вы не смогли бы создать два потока, выполняющих одну и ту же функцию, так как оба потока совместно использовали бы статическую переменную. Другое решение этой проблемы (и его более сложные варианты) базируется на методах синхронизации потоков, речь о которых пойдет в главах 8, 9 и 10.

Параметр *fdwCreate*

Этот параметр определяет дополнительные флаги, управляющие созданием потока. Он принимает одно из двух значений: 0 (исполнение потока начинается немедленно) или CREATE_SUSPENDED. В последнем случае система создает поток, инициализирует его и приостанавливает до последующих указаний.

Флаг CREATE_SUSPENDED позволяет программе изменить какие-либо свойства потока перед тем, как он начнет выполнять код. Правда, необходимость в этом возникает довольно редко. Одно из применений этого флага демонстрирует программа-пример JobLab из главы 5.

Параметр *pdwThreadID*

Последний параметр функции *CreateThread* — это адрес переменной типа DWORD, в которой функция возвращает идентификатор, приписанный системой новому потоку. (Идентификаторы процессов и потоков рассматривались в главе 4.)



В Windows 2000 и Windows NT 4 в этом параметре можно передавать NULL (обычно так и делается). Тем самым Вы сообщаете функции, что Вас не интересует идентификатор потока. Но в Windows 95/98 это приведет к ошибке, так как функция попытается записать идентификатор потока по нулевому адресу, что недопустимо. И поток не будет создан.

Такое несоответствие между операционными системами может создать разработчикам приложений массу проблем. Допустим, Вы пишете и тестируете программу в Windows 2000 (которая создает поток, даже если Вы передаете NULL в *pdwThreadID*). Но вот Вы запускаете приложение в Windows 98, и функция *CreateThread*, естественно, дает ошибку. Вывод один: тщательно тестируйте свое приложение во всех операционных системах, в которых оно будет работать.

Завершение потока

Поток можно завершить четырьмя способами:

- функция потока возвращает управление (рекомендуемый способ);
- поток самоуничтожается вызовом функции *ExitThread* (нежелательный способ);
- один из потоков данного или стороннего процесса вызывает функцию *TerminateThread* (нежелательный способ);
- завершается процесс, содержащий данный поток (тоже нежелательно).

В этом разделе мы обсудим перечисленные способы завершения потока, а также рассмотрим, что на самом деле происходит в момент его окончания.

Возврат управления функцией потока

Функцию потока следует проектировать так, чтобы поток завершался только после того, как она возвращает управление. Это единственный способ, гарантирующий корректную очистку всех ресурсов, принадлежавших Вашему потоку. При этом:

- любые C++-объекты, созданные данным потоком, уничтожаются соответствующими деструкторами;
- система корректно освобождает память, которую занимал стек потока;
- система устанавливает код завершения данного потока (поддерживаемый объектом ядра «поток») — его и возвращает Ваша функция потока;
- счетчик пользователей данного объекта ядра «поток» уменьшается на 1.

Функция *ExitThread*

Поток можно завершить принудительно, вызвав:

```
VOID ExitThread(DWORD dwExitCode);
```

При этом освобождаются все ресурсы операционной системы, выделенные данному потоку, но C/C++-ресурсы (например, объекты, созданные из C++-классов) не очищаются. Именно поэтому лучше возвращать управление из функции потока, чем самому вызывать функцию *ExitThread*. (Подробнее на эту тему см. раздел «Функция *ExitProcess*» в главе 4.)

В параметр *dwExitCode* Вы помещаете значение, которое система рассматривает как код завершения потока. Возвращаемого значения у этой функции нет, потому что после ее вызова поток перестает существовать.



ExitThread — это Windows-функция, которая уничтожает поток. Но никогда не вызывайте ее, если Вы пишете код на C/C++. Вместо нее Вы должны использовать функцию *_endthreadex* из библиотеки Visual C++. (Если Вы работаете с другим компилятором, он должен поддерживать свой эквивалент функции *ExitThread*.) Что именно делает *_endthreadex* и почему это так важно, я объясню потом.

Функция *TerminateThread*

Вызов этой функции также завершает поток:

```
BOOL TerminateThread(  
    HANDLE hThread,  
    DWORD dwExitCode);
```

В отличие от *ExitThread*, которая уничтожает только вызывающий поток, эта функция завершает поток, указанный в параметре *hThread*. В параметр *dwExitCode* Вы помещаете значение, которое система рассматривает как код завершения потока. После того как поток будет уничтожен, счетчик пользователей его объекта ядра «поток» уменьшится на 1.



TerminateThread — функция асинхронная, т. е. она сообщает системе, что Вы хотите завершить поток, но к тому времени, когда она вернет управление, поток может быть еще не уничтожен. Так что, если Вам нужно точно знать момент завершения потока, используйте *WaitForSingleObject* (см. главу 9) или аналогичную функцию, передав ей описатель этого потока.

Корректно написанное приложение не должно вызывать эту функцию, поскольку поток не получает никакого уведомления о завершении; из-за этого он не может выполнить должную очистку ресурсов.



Уничтожение потока при вызове *ExitThread* или возврате управления из функции потока приводит к разрушению его стека. Но если он завершен функцией *TerminateThread*, система не уничтожает стек, пока не завершится и процесс, которому принадлежал этот поток. Так сделано потому, что другие потоки могут использовать указатели, ссылающиеся на данные в стеке завершенного потока. Если бы они обратились к несуществующему стеку, произошло бы нарушение доступа.

Кроме того, при завершении потока система уведомляет об этом все DLL, подключенные к процессу — владельцу завершенного потока. Но при вызове *TerminateThread* такого не происходит, и процесс может быть завершен некорректно. (Подробнее на эту тему см. главу 20.)

Если завершается процесс

Функции *ExitProcess* и *TerminateProcess*, рассмотренные в главе 4, тоже завершают потоки. Единственное отличие в том, что они прекращают выполнение всех потоков, принадлежавших завершенному процессу. При этом гарантируется высвобождение любых выделенных процессу ресурсов, в том числе стеков потоков. Однако эти две функции уничтожают потоки принудительно — так, будто для каждого из них вызывается функция *TerminateThread*. А это означает, что очистка проводится некорректно: деструкторы C++-объектов не вызываются, данные на диск не сбрасываются и т. д.

Что происходит при завершении потока

А происходит вот что.

- Освобождаются все описатели User-объектов, принадлежавших потоку. В Windows большинство объектов принадлежит процессу, содержащему поток, из которого они были созданы. Сам поток владеет только двумя User-объектами: окнами и ловушками (hooks). Когда поток, создавший такие объекты, завершается, система уничтожает их автоматически. Прочие объекты разрушаются, только когда завершается владевший ими процесс.
- Код завершения потока меняется со `STILL_ACTIVE` на код, переданный в функцию *ExitThread* или *TerminateThread*.
- Объект ядра «поток» переводится в свободное состояние.
- Если данный поток является последним активным потоком в процессе, завершается и сам процесс.
- Счетчик пользователей объекта ядра «поток» уменьшается на 1.

При завершении потока сопоставленный с ним объект ядра «поток» не освобождается до тех пор, пока не будут закрыты все внешние ссылки на этот объект.

Когда поток завершился, толку от его описателя другим потокам в системе в общем немного. Единственное, что они могут сделать, — вызвать функцию *GetExitCodeThread*, проверить, завершен ли поток, идентифицируемый описателем *hThread*, и, если да, определить его код завершения.

```
BOOL GetExitCodeThread(
    HANDLE hThread,
    PDWORD pdwExitCode);
```

Код завершения возвращается в переменной типа DWORD, на которую указывает *pdwExitCode*. Если поток не завершен на момент вызова *GetExitCodeThread*, функция записывает в эту переменную идентификатор STILL_ACTIVE (0x103). При успешном вызове функция возвращает TRUE. К использованию описателя для определения факта завершения потока мы еще вернемся в главе 9.

Кое-что о внутреннем устройстве потока

Я уже объяснил Вам, как реализовать функцию потока и как заставить систему создать поток, который выполнит эту функцию. Теперь мы попробуем разобраться, как система справляется с данной задачей.

На рис. 6-1 показано, что именно должна сделать система, чтобы создать и инициализировать поток. Давайте приглядимся к этой схеме повнимательнее. Вызов *CreateThread* заставляет систему создать объект ядра «поток». При этом счетчику числа его пользователей присваивается начальное значение, равное 2. (Объект ядра «поток» уничтожается только после того, как прекращается выполнение потока и закрывается описатель, возвращенный функцией *CreateThread*.) Также инициализируются другие свойства этого объекта: счетчик числа простоев (suspension count) получает значение 1, а код завершения — значение STILL_ACTIVE (0x103). И, наконец, объект переводится в состояние «занято».

Создав объект ядра «поток», система выделяет стеку потока память из адресного пространства процесса и записывает в его самую верхнюю часть два значения. (Стеки потоков всегда строятся от старших адресов памяти к младшим.) Первое из них является значением параметра *pvParam*, переданного Вами функции *CreateThread*, а второе — это содержимое параметра *pfnStartAddr*, который Вы тоже передаете в *CreateThread*.

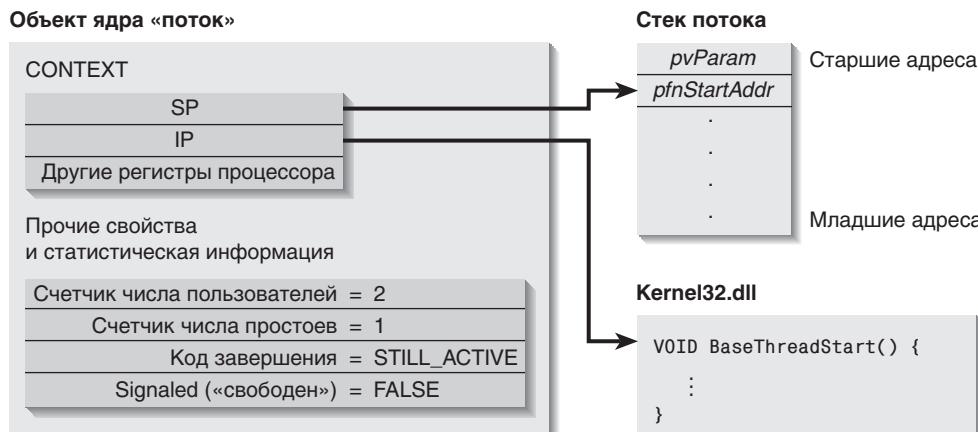


Рис. 6-1. Так создается и инициализируется поток

У каждого потока собственный набор регистров процессора, называемый *контекстом* потока. Контекст отражает состояние регистров процессора на момент последнего исполнения потока и записывается в структуру CONTEXT (она определена в заголовочном файле WinNT.h). Эта структура содержится в объекте ядра «поток».

Указатель команд (IP) и указатель стека (SP) — два самых важных регистра в контексте потока. Вспомните: потоки выполняются в контексте процесса. Соответственно эти регистры всегда указывают на адреса памяти в адресном пространстве процесса. Когда система инициализирует объект ядра «поток», указателю стека в структуре CONTEXT присваивается тот адрес, по которому в стек потока было записано значение *pfnStartAddr*, а указателю команд — адрес недокументированной (и неэкспортируемой) функции *BaseThreadStart*. Эта функция содержится в модуле Kernel32.dll, где, кстати, реализована и функция *CreateThread*.

Вот главное, что делает *BaseThreadStart*:

```
VOID BaseThreadStart(PTHREAD_START_ROUTINE pfnStartAddr, PVOID pvParam) {
    __try {
        ExitThread((pfnStartAddr)(pvParam));
    }
    __except(UnhandledExceptionFilter(GetExceptionInformation())) {
        ExitProcess(GetExceptionCode());
    }
    // ПРИМЕЧАНИЕ: мы никогда не попадем сюда
}
```

После инициализации потока система проверяет, был ли передан функции *CreateThread* флаг CREATE_SUSPENDED. Если нет, система обнуляет его счетчик числа простоев, и потоку может быть выделено процессорное время. Далее система загружает в регистры процессора значения, сохраненные в контексте потока. С этого момента поток может выполнять код и манипулировать данными в адресном пространстве своего процесса.

Поскольку указатель команд нового потока установлен на *BaseThreadStart*, именно с этой функции и начнется выполнение потока. Глядя на ее прототип, можно подумать, будто *BaseThreadStart* передаются два параметра, а значит, она вызывается из какой-то другой функции, но это не так. Новый поток просто начинает с нее свою работу. *BaseThreadStart* получает доступ к двум параметрам, которые появляются у нее потому, что операционная система записывает соответствующие значения в стек потока (а через него параметры как раз и передаются функциям). Правда, на некоторых аппаратных платформах параметры передаются не через стек, а с использованием определенных регистров процессора. Поэтому на таких аппаратных платформах система — прежде чем разрешить потоку выполнение функции *BaseThreadStart* — инициализирует нужные регистры процессора.

Когда новый поток выполняет *BaseThreadStart*, происходит следующее.

- Ваша функция потока включается во фрейм структурной обработки исключений (далее для краткости — SEH-фрейм), благодаря чему любое исключение, если оно происходит в момент выполнения Вашего потока, получает хоть какую-то обработку, предлагаемую системой по умолчанию. Подробнее о структурной обработке исключений см. главы 23, 24 и 25.
- Система обращается к Вашей функции потока, передавая ей параметр *pvParam*, который Вы ранее передали функции *CreateThread*.
- Когда Ваша функция потока возвращает управление, *BaseThreadStart* вызывает *ExitThread*, передавая ей значение, возвращенное Вашей функцией. Счетчик числа пользователей объекта ядра «поток» уменьшается на 1, и выполнение потока прекращается.

- Если Ваш поток вызывает необрабатываемое им исключение, его обрабатывает SEH-фрейм, построенный функцией *BaseThreadStart*. Обычно в результате этого появляется окно с каким-нибудь сообщением, и, когда пользователь закрывает его, *BaseThreadStart* вызывает *ExitProcess* и завершает весь процесс, а не только тот поток, в котором произошло исключение.

Обратите внимание, что из *BaseThreadStart* поток вызывает либо *ExitThread*, либо *ExitProcess*. А это означает, что поток никогда не выходит из данной функции; он всегда уничтожается внутри нее. Вот почему у *BaseThreadStart* нет возвращаемого значения — она просто ничего не возвращает.

Кстати, именно благодаря *BaseThreadStart* Ваша функция потока получает возможность вернуть управление по окончании своей работы. *BaseThreadStart*, вызывая функцию потока, заталкивает в стек свой адрес возврата и тем самым сообщает ей, куда надо вернуться. Но сама *BaseThreadStart* не возвращает управление. Иначе возникло бы нарушение доступа, так как в стеке потока нет ее адреса возврата.

При инициализации первичного потока его указатель команд устанавливается на другую недокументированную функцию — *BaseProcessStart*. Она почти идентична *BaseThreadStart* и выглядит примерно так:

```
VOID BaseProcessStart(PPROCESS_START_ROUTINE pfnStartAddr) {  
    __try {  
        ExitThread((pfnStartAddr)());  
    }  
    __except(UnhandledExceptionFilter(GetExceptionInformation())) {  
        ExitProcess(GetExceptionCode());  
    }  
    // ПРИМЕЧАНИЕ: мы никогда не попадем сюда  
}
```

Единственное различие между этими функциями в отсутствии ссылки на параметр *pvParam*. Функция *BaseProcessStart* обращается к стартовому коду библиотеки C/C++, который выполняет необходимую инициализацию, а затем вызывает Вашу входную функцию *main*, *wmain*, *WinMain* или *wWinMain*. Когда входная функция возвращает управление, стартовый код библиотеки C/C++ вызывает *ExitProcess*. Поэтому первичный поток приложения, написанного на C/C++, никогда не возвращается в *BaseProcessStart*.

Некоторые соображения по библиотеке C/C++

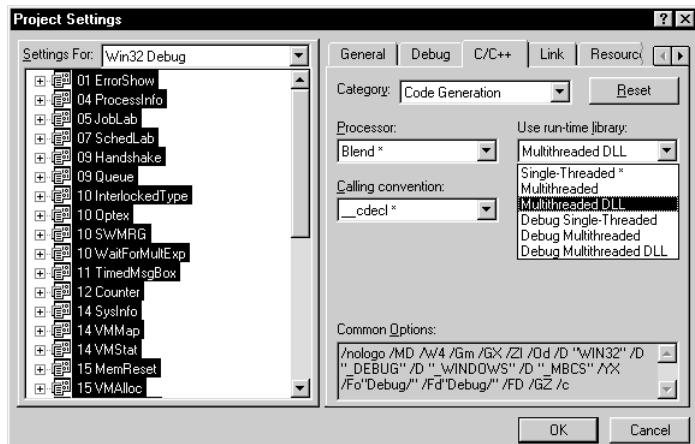
Microsoft поставляет с Visual C++ шесть библиотек C/C++. Их краткое описание представлено в следующей таблице.

Имя библиотеки	Описание
LibC.lib	Статически подключаемая библиотека для однопоточных приложений (используется по умолчанию при создании нового проекта)
LibCD.lib	Отладочная версия статически подключаемой библиотеки для однопоточных приложений
LibCMt.lib	Статически подключаемая библиотека для многопоточных приложений
LibCMtD.lib	Отладочная версия статически подключаемой библиотеки для многопоточных приложений
MSVCRT.lib	Библиотека импорта для динамического подключения рабочей версии MSVCRT.dll; поддерживает как одно-, так и многопоточные приложения

продолжение

Имя библиотеки	Описание
MSVCRTD.lib	Библиотека импорта для динамического подключения отладочной версии MSVCRTD.dll; поддерживает как одно-, так и многопоточные приложения

При реализации любого проекта нужно знать, с какой библиотекой его следует связать. Конкретную библиотеку можно выбрать в диалоговом окне Project Settings: на вкладке C/C++ в списке Category укажите Code Generation, а в списке Use Run-Time Library — одну из шести библиотек.



Наверное, Вам уже хочется спросить: «А зачем мне отдельные библиотеки для однопоточных и многопоточных программ?» Отвечаю. Стандартная библиотека C была разработана где-то в 1970 году — задолго до появления самого понятия многопоточности. Авторы этой библиотеки, само собой, не задумывались о проблемах, связанных с многопоточными приложениями.

Возьмем, к примеру, глобальную переменную *errno* из стандартной библиотеки C. Некоторые функции, если происходит какая-нибудь ошибка, записывают в эту переменную соответствующий код. Допустим, у Вас есть такой фрагмент кода:

```
BOOL fFailure = (system("NOTE PAD. EXE README. TXT") == -1);

if (fFailure) {
    switch (errno) {
        case E2BIG:           // список аргументов или размер окружения слишком велик
            break;
        case ENOENT:          // командный интерпретатор не найден
            break;
        case ENOEXEC:          // неверный формат командного интерпретатора
            break;
        case ENOMEM:          // недостаточно памяти для выполнения команды
            break;
    }
}
```

Теперь представим, что поток, выполняющий показанный выше код, прерван после вызова функции *system* и до оператора *if*. Допустим также, поток прерван для выпол-

нения другого потока (в том же процессе), который обращается к одной из функций библиотеки C, и та тоже заносит какое-то значение в глобальную переменную *errno*. Смотрите, что получается: когда процессор вернется к выполнению первого потока, в переменной *errno* окажется вовсе не то значение, которое было записано функцией *system*. Поэтому для решения этой проблемы нужно закрепить за каждым потоком свою переменную *errno*. Кроме того, понадобится какой-то механизм, который позволит каждому потоку ссыльаться на свою переменную *errno* и не трогать чужую.

Это лишь один пример того, что стандартная библиотека C/C++ не рассчитана на многопоточные приложения. Кроме *errno*, в ней есть еще целый ряд переменных и функций, с которыми возможны проблемы в многопоточной среде: *_doserrno*, *strtok*, *_wcstok*, *strerror*, *_strerror*, *tmpnam*, *tmpfile*, *asctime*, *_wasctime*, *gmtime*, *_ecvt*, *_fcvt* — список можно продолжить.

Чтобы многопоточные программы, использующие библиотеку C/C++, работали корректно, требуется создать специальную структуру данных и связать ее с каждым потоком, из которого вызываются библиотечные функции. Более того, они должны знать, что, когда Вы к ним обращаетесь, нужно просматривать этот блок данных в вызывающем потоке, чтобы не повредить данные в каком-нибудь другом потоке.

Так откуда же система знает, что при создании нового потока надо создать и этот блок данных? Ответ очень прост: не знает и знать не хочет. Вся ответственность — исключительно на Вас. Если Вы пользуетесь небезопасными в многопоточной среде функциями, то должны создавать потоки библиотечной функцией *_beginthreadex*, а не Windows-функцией *CreateThread*:

```
unsigned long _beginthreadex(
    void *security,
    unsigned stack_size,
    unsigned (*start_address)(void *),
    void *arglist,
    unsigned initflag,
    unsigned *thrdaddr);
```

У функции *_beginthreadex* тот же список параметров, что и у *CreateThread*, но их имена и типы несколько отличаются. (Группа, которая отвечает в Microsoft за разработку и поддержку библиотеки C/C++, считает, что библиотечные функции не должны зависеть от типов данных Windows.) Как и *CreateThread*, функция *_beginthreadex* возвращает описатель только что созданного потока. Поэтому, если Вы раньше пользовались функцией *CreateThread*, ее вызовы в исходном коде несложно заменить на вызовы *_beginthreadex*. Однако из-за некоторого расхождения в типах данных Вам придется позаботиться об их приведении к тем, которые нужны функции *_beginthreadex*, и тогда компилятор будет счастлив. Лично я создал небольшой макрос, *chBEGINTHREADEX*, который и делает всю эту работу в исходном коде.

```
typedef unsigned (_ _stdcall *PTHREAD_START) (void *);

#define chBEGINTHREADEX(psa, cbStack, pfnStartAddr, \
    pvParam, fdwCreate, pdwThreadID) \
        ((HANDLE) _beginthreadex( \
            (void *) (psa), \
            (unsigned) (cbStack), \
            (PTHREAD_START) (pfnStartAddr), \
            (void *) (pvParam), \
            (unsigned) (fdwCreate), \
            (unsigned *) (pdwThreadID)))
```

Заметьте, что функция `_beginthreadex` существует только в многопоточных версиях библиотеки C/C++. Связав проект с однопоточной библиотекой, Вы получите от компоновщика сообщение об ошибке «unresolved external symbol». Конечно, это сделано специально, потому что однопоточная библиотека не может корректно работать в многопоточном приложении. Также обратите внимание на то, что при создании нового проекта Visual Studio по умолчанию выбирает однопоточную библиотеку. Этот вариант не самый безопасный, и для многопоточных приложений Вы должны сами выбрать одну из многопоточных версий библиотеки C/C++.

Поскольку Microsoft поставляет исходный код библиотеки C/C++, несложно разобраться в том, что такого делает `_beginthreadex`, чего не делает `CreateThread`. На дистрибутивном компакт-диске Visual Studio ее исходный код содержится в файле Threadex.c. Чтобы не перепечатывать весь код, я решил дать Вам ее версию в псевдокоде, выделив самые интересные места.

```

unsigned long __cdecl _beginthreadex (
    void *psa,
    unsigned cbStack,
    unsigned (_stdcall * pfnStartAddr) (void *),
    void * pvParam,
    unsigned fdwCreate,
    unsigned *pdwThreadID) {

    _ptiddata ptd;           // указатель на блок данных потока
    unsigned long thdl;      // описатель потока

    // выделяется блок данных для нового потока
    if ((ptd = _calloc_crt(1, sizeof(struct tiddata))) == NULL)
        goto error_return;

    // инициализация блока данных
    initptd(ptd);

    // здесь запоминается нужная функция потока и параметр,
    // который мы хотим поместить в блок данных
    ptd->_initaddr = (void *) pfnStartAddr;
    ptd->_initarg = pvParam;

    // создание нового потока
    thdl = (unsigned long) CreateThread(psa, cbStack,
        _threadstartex, (PVOID) ptd, fdwCreate, pdwThreadID);
    if (thdl == NULL) {
        // создать поток не удалось; проводится очистка и сообщается об ошибке
        goto error_return;
    }

    // поток успешно создан; возвращается его описатель
    return(thdl);

error_return:
    // ошибка: не удалось создать блок данных или сам поток
    _free_crt(ptd);
    return((unsigned long)0L);
}

```

Несколько важных моментов, связанных с `_beginthreadex`.

- Каждый поток получает свой блок памяти `tidata`, выделяемый из кучи, которая принадлежит библиотеке C/C++. (Структура `tidata` определена в файле Mtdll.h. Она довольно любопытна, и я привел ее на рис. 6-2.)
- Адрес функции потока, переданный `_beginthreadex`, запоминается в блоке памяти `tidata`. Там же сохраняется и параметр, который должен быть передан этой функции.
- Функция `_beginthreadex` вызывает `CreateThread`, так как лишь с ее помощью операционная система может создать новый поток.
- При вызове `CreateThread` сообщается, что она должна начать выполнение нового потока с функции `_threadstartex`, а не с того адреса, на который указывает `pfnStartAddr`. Кроме того, функции потока передается не параметр `pvParam`, а адрес структуры `tidata`.
- Если все проходит успешно, `_beginthreadex`, как и `CreateThread`, возвращает описатель потока. В ином случае возвращается NULL.

```
struct _tidata {  
    unsigned long    _tid;           /* идентификатор потока */  
    unsigned long    _thandle;        /* описатель потока */  
    int              _terrno;         /* значение errno */  
    unsigned long    _tdoserrno;      /* значение _doserrno */  
    unsigned int     _fpds;           /* сегмент данных Floating Point */  
    unsigned long    _holdrand;       /* зародышевое значение для rand() */  
    char *           _token;          /* указатель (ptr) на метку strtok() */  
#ifdef _WIN32  
    wchar_t *        _wtoken;         /* ptr на метку wcstok() */  
#endif /* _WIN32 */  
    unsigned char *   _mtoken;         /* ptr на метку _mbstok() */  
  
    /* следующие указатели обрабатываются функцией malloc в период выполнения */  
    char *           _errmsg;         /* ptr на буфер strerror()/_strerror() */  
    char *           _namebuf0;        /* ptr на буфер tmpnam() */  
#ifdef _WIN32  
    wchar_t *        _wnamebuf0;       /* ptr на буфер _wttmpnam() */  
#endif /* _WIN32 */  
    char *           _namebuf1;        /* ptr на буфер tmpfile() */  
#ifdef _WIN32  
    wchar_t *        _wnamebuf1;       /* ptr на буфер _wttmpfile() */  
#endif /* _WIN32 */  
    char *           _asctimebuf;      /* ptr на буфер asctime() */  
#ifdef _WIN32  
    wchar_t *        _wasctimebuf;     /* ptr на буфер _wasctime() */  
#endif /* _WIN32 */  
    void *           _gmtimebuf;       /* ptr на структуру gmtime() */  
    char *           _cvtbuf;          /* ptr на буфер ecvt()/fcvt */  
  
    /* следующие поля используются кодом _beginthread */  
    void *           _initaddr;        /* начальный адрес пользовательского потока */  
    void *           _initarg;         /* начальный аргумент пользовательского потока */
```

Рис. 6-2. Локальная структура `tidata` потока, определенная в библиотеке C/C++

Рис. 6-2. продолжение

```

/* следующие три поля нужны для поддержки функции signal и обработки ошибок,
 * возникающих в период выполнения */
void *      _pxcptacttab;    /* ptr на таблицу "исключение-действие" */
void *      _tpxcptinfoptrs; /* ptr на указатели к информации об исключении */
int         _tfpecode;       /* код исключения для операций над числами
                                * с плавающей точкой */

/* следующее поле нужно подпрограммам NLG */
unsigned long _NLG_dwCode;

/*
 * данные для отдельного потока, используемые при обработке исключений в C++
 */
void *      _terminate;      /* подпрограмма terminate() */
void *      _unexpected;     /* подпрограмма unexpected() */
void *      _translator;     /* транслятор S.E. */
void *      _curexception;   /* текущее исключение */
void *      _curcontext;     /* контекст текущего исключения */
#if defined (_M_MRX000)
void *      _pFrameInfoChain;
void *      _pUnwindContext;
void *      _pExitContext;
int         _MipsPtdDelta;
int         _MipsPtdEpsilon;
#elif defined (_M_PPC)
void *      _pExitContext;
void *      _pUnwindContext;
void *      _pFrameInfoChain;
int         _FrameInfo[6];
#endif /* defined (_M_PPC) */
};

typedef struct _tiddata * _ptiddata;

```

Выяснив, как создается и инициализируется структура *tiddata* для нового потока, посмотрим, как она сопоставляется с этим потоком. Взгляните на исходный код функции *_threadstartex* (который тоже содержится в файле Threadex.c библиотеки C/C++). Вот моя версия этой функции в псевдокоде:

```

static unsigned long WINAPI threadstartex (void* ptd) {
    // Примечание: ptd - это адрес блока tiddata данного потока

    // блок tiddata сопоставляется с данным потоком
    TlsSetValue(_tlsindex, ptd);

    // идентификатор этого потока записывается в tiddata
    ((_ptiddata) ptd)->_tid = GetCurrentThreadId();

    // здесь инициализируется поддержка операций над числами с плавающей точкой
    // (код не показан)

```

см. след. стр.

```
// пользовательская функция потока включается в SEH-фрейм для обработки
// ошибок периода выполнения и поддержки signal
__try {
    // здесь вызывается функция потока, которой передается нужный параметр;
    // код завершения потока передается _endthreadex
    _endthreadex(
        ( (unsigned (WINAPI * )(void * ))((_ptidata)ptd)->_initaddr) )
        ( ((_ptidata)ptd)->_initarg ) );
}

__except(_XcptFilter(GetExceptionCode(), GetExceptionInformation())){
    // обработчик исключений из библиотеки С не даст нам попасть сюда
    _exit(GetExceptionCode());
}

// здесь мы тоже никогда не будем, так как в этой функции поток умирает
return(0L);
}
```

Несколько важных моментов, связанных со *_threadstartex*.

- Новый поток начинает выполнение с *BaseThreadStart* (в Kernel32.dll), а затем переходит в *_threadstartex*.
- В качестве единственного параметра функции *_threadstartex* передается адрес блока *tidata* нового потока.
- Windows-функция *TlsSetValue* сопоставляет с вызывающим потоком значение, которое называется локальной памятью потока (Thread Local Storage, TLS) (о ней я расскажу в главе 21), а *_threadstartex* сопоставляет блок *tidata* с новым потоком.
- Функция потока заключается в SEH-фрейм. Он предназначен для обработки ошибок периода выполнения (например, не перехваченных исключений C++), поддержки библиотечной функции *signal* и др. Этот момент, кстати, очень важен. Если бы Вы создали поток с помощью *CreateThread*, а потом вызвали библиотечную функцию *signal*, она работала бы некорректно.
- Далее вызывается функция потока, которой передается нужный параметр. Адрес этой функции и ее параметр были сохранены в блоке *tidata* функцией *_beginthreadex*.
- Значение, возвращаемое функцией потока, считается кодом завершения этого потока. Обратите внимание: *_threadstartex* не возвращается в *BaseThreadStart*. Иначе после уничтожения потока его блок *tidata* так и остался бы в памяти. А это привело бы к утечке памяти в Вашем приложении. Чтобы избежать этого, *_threadstartex* вызывает другую библиотечную функцию, *_endthreadex*, и передает ей код завершения.

Последняя функция, которую нам нужно рассмотреть, — это *_endthreadex* (ее исходный код тоже содержится в файле Threadex.c). Вот как она выглядит в моей версии (в псевдокоде):

```
void __cdecl _endthreadex (unsigned retcode) {
    _ptidata ptd;      // указатель на блок данных потока

    // здесь проводится очистка ресурсов, выделенных для поддержки операций
    // над числами с плавающей точкой (код не показан)
```

```

// определение адреса блока tiddata данного потока
ptd = _getptd();

// высвобождение блока tiddata
_freeptd(ptd);

// завершение потока
ExitThread(retcode);
}

```

Несколько важных моментов, связанных с *_endthreadex*.

- Библиотечная функция *_getptd* обращается к Windows-функции *TlsGetValue*, которая сообщает адрес блока памяти *tiddata* вызывающего потока.
- Этот блок освобождается, и вызовом *ExitThread* поток разрушается. При этом, конечно, передается корректный код завершения.

Где-то в начале главы я уже говорил, что прямого обращения к функции *ExitThread* следует избегать. Это правда, и я не отказываюсь от своих слов. Тогда же я сказал, что это приводит к уничтожению вызывающего потока и не позволяет ему вернуться из выполняемой в данный момент функции. А поскольку она не возвращает управление, любые созданные Вами C++-объекты не разрушаются. Так вот, теперь у Вас есть еще одна причина не вызывать *ExitThread*: она не дает освободить блок памяти *tiddata* потока, из-за чего в Вашем приложении может наблюдаться утечка памяти (до его завершения).

Разработчики Microsoft Visual C++, конечно, прекрасно понимают, что многие все равно будут пользоваться функцией *ExitThread*, поэтому они кое-что сделали, чтобы свести к минимуму вероятность утечки памяти. Если Вы действительно так хотите самостоятельно уничтожить свой поток, можете вызвать из него *_endthreadex* (вместо *ExitThread*) и тем самым освободить его блок *tiddata*. И все же я не рекомендую этого.

Сейчас Вы уже должны понимать, зачем библиотечным функциям нужен отдельный блок данных для каждого порожденного потока и каким образом после вызова *_beginthreadex* происходит создание и инициализация этого блока данных, а также его связывание с только что созданным потоком. Кроме того, Вы уже должны разбираться в том, как функция *_endthreadex* освобождает этот блок по завершении потока.

Как только блок данных инициализирован и сопоставлен с конкретным потоком, любая библиотечная функция, к которой обращается поток, может легко узнать адрес его блока и таким образом получить доступ к данным, принадлежащим этому потоку.

Ладно, с функциями все ясно, теперь попробуем проследить, что происходит с глобальными переменными вроде *errno*. В заголовочных файлах С эта переменная определена так:

```

#if defined(_MT) || defined(_DLL)
extern int * __cdecl _errno(void);
#define errno (*_errno())
#else /* !defined(_MT) && !defined(_DLL) */
extern int errno;
#endif /* _MT || _DLL */

```

Создавая многопоточное приложение, надо указывать в командной строке компилятора один из ключей: /MT (многопоточное приложение) или /MD (многопоточ-

ная DLL); тогда компилятор определит идентификатор _MT. После этого, ссылаясь на *errno*, Вы будете на самом деле вызывать внутреннюю функцию *_errno* из библиотеки C/C++. Она возвращает адрес элемента данных *errno* в блоке, сопоставленном с вызывающим потоком. Кстати, макрос *errno* составлен так, что позволяет получать содержимое памяти по этому адресу. А сделано это для того, чтобы можно было писать, например, такой код:

```
int *p = &errno;
if (*p == ENOMEM) {
    :
}
```

Если бы внутренняя функция *_errno* просто возвращала значение *errno*, этот код не удалось бы скомпилировать.

Многопоточная версия библиотеки C/C++, кроме того, «обертывает» некоторые функции синхронизирующими примитивами. Ведь если бы два потока одновременно вызывали функцию *malloc*, куча могла бы быть повреждена. Поэтому в многопоточной версии библиотеки потоки не могут одновременно выделять память из кучи. Второй поток она заставляет ждать до тех пор, пока первый не выйдет из функции *malloc*, и лишь тогда второй поток получает доступ к *malloc*. (Подробнее о синхронизации потоков мы поговорим в главах 8, 9 и 10.)

Конечно, все эти дополнительные операции не могли не отразиться на быстродействии многопоточной версии библиотеки. Поэтому Microsoft, кроме многопоточной, поставляет и однопоточную версию статически подключаемой библиотеки C/C++.

Динамически подключаемая версия библиотеки C/C++ вполне универсальна: ее могут использовать любые выполняемые приложения и DLL, которые обращаются к библиотечным функциям. По этой причине данная библиотека существует лишь в многопоточной версии. Поскольку она поставляется в виде DLL, ее код не нужно включать непосредственно в EXE- и DLL-модули, что существенно уменьшает их размер. Кроме того, если Microsoft исправляет какую-то ошибку в такой библиотеке, то и программы, построенные на ее основе, автоматически избавляются от этой ошибки.

Как Вы, наверное, и предполагали, стартовый код из библиотеки C/C++ создает и инициализирует блок данных для первичного потока приложения. Это позволяет без всяких опасений вызывать из первичного потока любые библиотечные функции. А когда первичный поток заканчивает выполнение своей входной функции, блок данных завершающего потока освобождается самой библиотекой. Более того, стартовый код делает все необходимое для структурной обработки исключений, благодаря чему из первичного потока можно спокойно обращаться и к библиотечной функции *signal*.

Ой, вместо *_beginthreadex* я по ошибке вызвал *CreateThread*

Вас, наверное, интересует, что случится, если создать поток не библиотечной функцией *_beginthreadex*, а Windows-функцией *CreateThread*. Когда этот поток вызовет какую-нибудь библиотечную функцию, которая манипулирует со структурой *tiddata*, произойдет следующее. (Большинство библиотечных функций реентерабельно и не требует этой структуры.) Сначала эта функция попытается выяснить адрес блока данных потока (вызовом *TlsGetValue*). Получив NULL вместо адреса *tiddata*, она узнает, что вызывающий поток не сопоставлен с таким блоком. Тогда библиотечная функция тут

же создаст и инициализирует блок *tiddata* для вызывающего потока. Далее этот блок будет сопоставлен с потоком (через *TlsSetValue*) и останется при нем до тех пор, пока выполнение потока не прекратится. С этого момента данная функция (как, впрочем, и любая другая из библиотеки C/C++) сможет пользоваться блоком *tiddata* потока.

Как это ни фантастично, но Ваш поток будет работать почти без глюков. Хотя некоторые проблемы все же появятся. Во-первых, если этот поток воспользуется библиотечной функцией *signal*, весь процесс завершится, так как SEH-фрейм не подготовлен. Во-вторых, если поток завершится, не вызвав *_endthreadex*, его блок данных не высвободится и произойдет утечка памяти. (Да и кто, интересно, вызовет *_endthreadex* из потока, созданного с помощью *CreateThread*?)



Если Вы связываете свой модуль с многопоточной DLL-версией библиотеки C/C++, то при завершении потока и высвобождении блока *tiddata* (если он был создан), библиотека получает уведомление DLL_THREAD_DETACH. Даже несмотря на то что это предотвращает утечку памяти, связанную с блоком *tiddata*, я настоятельно советую создавать потоки через *_beginthreadex*, а не с помощью *CreateThread*.

Библиотечные функции, которые лучше не вызывать

В библиотеке C/C++ содержится две функции:

```
unsigned long _beginthread(
    void (*__cdecl *start_address)(void *),
    unsigned stack_size,
    void *arglist);

и

void _endthread(void);
```

Первоначально они были созданы для того, чем теперь занимаются новые функции *_beginthreadex* и *_endthreadex*. Но, как видите, у *_beginthread* параметров меньше, и, следовательно, ее возможности ограничены в сравнении с полнофункциональной *_beginthreadex*. Например, работая с *_beginthread*, нельзя создать поток с атрибутами защиты, отличными от присваиваемых по умолчанию, нельзя создать поток и тут же его задержать — нельзя даже получить идентификатор потока. С функцией *_endthread* та же история: она не принимает никаких параметров, а это значит, что по окончании работы потока его код завершения всегда равен 0.

Однако с функцией *_endthread* дело обстоит куда хуже, чем кажется: перед вызовом *ExitThread* она обращается к *CloseHandle* и передает ей описатель нового потока. Чтобы разобраться, в чем тут проблема, взгляните на следующий код:

```
DWORD dwExitCode;
HANDLE hThread = _beginthread(...);
GetExitCodeThread(hThread, &dwExitCode);
CloseHandle(hThread);
```

Весьма вероятно, что созданный поток отработает и завершится еще до того, как первый поток успеет вызвать функцию *GetExitCodeThread*. Если так и случится, значение в *hThread* окажется недействительным, потому что *_endthread* уже закрыла описатель нового потока. И, естественно, вызов *CloseHandle* даст ошибку.

Новая функция `_endthreadex` не закрывает описатель потока, поэтому фрагмент кода, приведенный выше, будет нормально работать (если мы, конечно, заменим вызов `_beginthread` на вызов `_beginthreadex`). И в заключение, напомню еще раз: как только функция потока возвращает управление, `_beginthreadex` самостоятельно вызывает `_endthreadex`, а `_beginthread` обращается к `_endthread`.

Как узнать о себе

Потоки часто обращаются к Windows-функциям, которые меняют среду выполнения. Например, потоку может понадобиться изменить свой приоритет или приоритет процесса. (Приоритеты рассматриваются в главе 7.) И поскольку это не редкость, когда поток модифицирует среду (собственную или процесса), в Windows предусмотрены функции, позволяющие легко ссылаться на объекты ядра текущего процесса и потока:

```
HANDLE GetCurrentProcess();
HANDLE GetCurrentThread();
```

Обе эти функции возвращают псевдоописатель объекта ядра «процесс» или «поток». Они не создают новые описатели в таблице описателей, которая принадлежит вызывающему процессу, и не влияют на счетчики числа пользователей объектов ядра «процесс» и «поток». Поэтому, если вызвать `CloseHandle` и передать ей псевдоописатель, она проигнорирует вызов и просто вернет FALSE.

Псевдоописатели можно использовать при вызове функций, которым нужен описатель процесса. Так, поток может запросить все временные показатели своего процесса, вызвав `GetProcessTimes`:

```
FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
GetProcessTimes(GetCurrentProcess(),
    &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
```

Аналогичным образом поток может выяснить собственные временные показатели, вызвав `GetThreadTimes`:

```
FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
GetThreadTimes(GetCurrentThread(),
    &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
```

Некоторые Windows-функции позволяют указывать конкретный процесс или поток по его уникальному в рамках всей системы идентификатору. Вот функции, с помощью которых поток может выяснить такой идентификатор — собственный или своего процесса:

```
DWORD GetCurrentProcessId();
DWORD GetCurrentThreadId();
```

По сравнению с функциями, которые возвращают псевдоописатели, эти функции, как правило, не столь полезны, но когда-то и они могут пригодиться.

Преобразование псевдоописателя в настоящий описатель

Иногда бывает нужно выяснить настоящий, а не псевдоописатель потока. Под «настоящим» я подразумеваю описатель, который однозначно идентифицирует уникальный поток. Вдумайтесь в такой фрагмент кода:

```

DWORD WINAPI ParentThread(PVOID pvParam) {
    HANDLE hThreadParent = GetCurrentThread();
    CreateThread(NULL, 0, ChildThread, (PVOID) hThreadParent, 0, NULL);
    // далее следует какой-то код...
}

DWORD WINAPI ChildThread(PVOID pvParam) {
    HANDLE hThreadParent = (HANDLE) pvParam;
    FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
    GetThreadTimes(hThreadParent,
        &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
    // далее следует какой-то код...
}

```

Вы заметили, что здесь не все ладно? Идея была в том, чтобы родительский поток передавал дочернему свой описатель. Но он передает псевдо-, а не настоящий описатель. Начиная выполнение, дочерний поток передает этот псевдоописатель функции *GetThreadTimes*, и она вследствие этого возвращает временные показатели своего — а вовсе не родительского! — потока. Происходит так потому, что псевдоописатель является описателем текущего потока, т. е. того, который вызывает эту функцию.

Чтобы исправить приведенный выше фрагмент кода, превратим псевдоописатель в настоящий через функцию *DuplicateHandle* (о ней я рассказывал в главе 3):

```

BOOL DuplicateHandle(
    HANDLE hSourceProcess,
    HANDLE hSource,
    HANDLE hTargetProcess,
    PHANDLE phTarget,
    DWORD fdwAccess,
    BOOL bInheritHandle,
    DWORD fdwOptions);

```

Обычно она используется для создания нового «процессо-зависимого» описателя из описателя объекта ядра, значение которого увязано с другим процессом. А мы воспользуемся *DuplicateHandle* не совсем по назначению и скорректируем с ее помощью наш фрагмент кода так:

```

DWORD WINAPI ParentThread(PVOID pvParam) {
    HANDLE hThreadParent;

    DuplicateHandle(
        GetCurrentProcess(),           // описатель процесса, к которому
                                       // относится псевдоописатель потока;
        GetCurrentThread(),           // псевдоописатель родительского потока;
        GetCurrentProcess(),           // описатель процесса, к которому
                                       // относится новый, настоящий
                                       // описатель потока;
        &hThreadParent,               // даст новый, настоящий описатель,
                                       // идентифицирующий родительский поток;
        0,                           // игнорируется из-за DUPLICATE_SAME_ACCESS;
        FALSE,                      // новый описатель потока ненаследуемый;
        DUPLICATE_SAME_ACCESS);     // новому описателю потока присваиваются
                                       // те же атрибуты защиты, что и псевдоописателю

```

см. след. стр.

```
CreateThread(NULL, 0, ChildThread, (PVOID) hThreadParent, 0, NULL);
// далее следует какой-то код...
}

DWORD WINAPI ChildThread(PVOID pvParam) {
    HANDLE hThreadParent = (HANDLE) pvParam;
    FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
    GetThreadTimes(hThreadParent,
        &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
    CloseHandle(hThreadParent);
    // далее следует какой-то код...
}
```

Теперь родительский поток преобразует свой «двусмысленный» псевдоописатель в настоящий описатель, однозначно определяющий родительский поток, и передает его в *CreateThread*. Когда дочерний поток начинает выполнение, его параметр *pvParam* содержит настоящий описатель потока. В итоге вызов какой-либо функции с этим описателем влияет не на дочерний, а на родительский поток.

Поскольку *DuplicateHandle* увеличивает счетчик пользователей указанного объекта ядра, то, закончив работу с продублированным описателем объекта, очень важно не забыть уменьшить счетчик. Сразу после обращения к *GetThreadTimes* дочерний поток вызывает *CloseHandle*, уменьшая тем самым счетчик пользователей объекта «родительский поток» на 1. В этом фрагменте кода я исходил из того, что дочерний поток не вызывает других функций с передачей этого описателя. Если же ему надо вызвать какие-то функции с передачей описателя родительского потока, то, естественно, к *CloseHandle* следует обращаться только после того, как необходимость в этом описателе у дочернего потока отпадет.

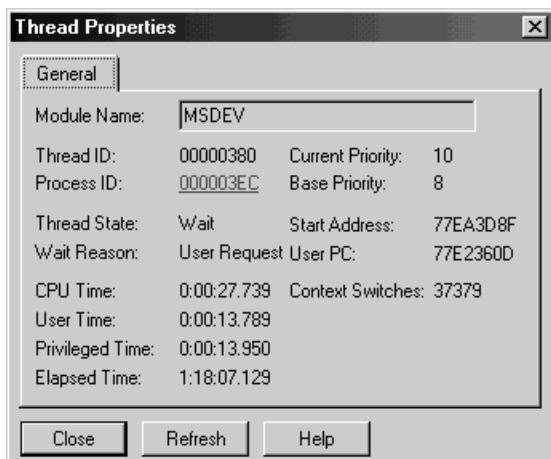
Надо заметить, что *DuplicateHandle* позволяет преобразовать и псевдоописатель процесса. Вот как это сделать:

```
HANDLE hProcess;
DuplicateHandle(
    GetCurrentProcess(),           // описатель процесса, к которому
                                    // относится псевдоописатель;
    GetCurrentProcess(),           // псевдоописатель процесса;
    GetCurrentProcess(),           // описатель процесса, к которому
                                    // относится новый, настоящий описатель;
    &hProcess,                   // даст новый, настоящий описатель,
                                // идентифицирующий процесс;
    0,                          // игнорируется из-за DUPLICATE_SAME_ACCESS;
    FALSE,                      // новый описатель процесса ненаследуемый;
    DUPLICATE_SAME_ACCESS);      // новому описателю процесса присваиваются
                                // те же атрибуты защиты, что и псевдоописателю
```

Планирование потоков, приоритет и привязка к процессорам

Операционная система с вытесняющей многозадачностью должна использовать тот или иной алгоритм, позволяющий ей распределять процессорное время между потоками. Здесь мы рассмотрим алгоритмы, применяемые в Windows 98 и Windows 2000.

В главе 6 мы уже обсудили структуру CONTEXT, поддерживаемую в объекте ядра «поток», и выяснили, что она отражает состояние регистров процессора на момент последнего выполнения потока процессором. Каждые 20 мс (или около того) Windows просматривает все существующие объекты ядра «поток» и отмечает те из них, которые могут получать процессорное время. Далее она выбирает один из таких объектов и загружает в регистры процессора значения из его контекста. Эта операция называется *переключением контекста* (context switching). По каждому потоку Windows ведет учет того, сколько раз он подключался к процессору. Этот показатель сообщают специальные утилиты вроде Microsoft Spy++. Например, на иллюстрации ниже показан список свойств одного из потоков. Обратите внимание, что этот поток подключался к процессору 37379 раз.



Поток выполняет код и манипулирует данными в адресном пространстве своего процесса. Примерно через 20 мс Windows сохранит значения регистров процессора в контексте потока и приостановит его выполнение. Далее система просмотрит остальные объекты ядра «поток», подлежащие выполнению, выберет один из них, загрузит его контекст в регистры процессора, и все повторится. Этот цикл операций — выбор потока, загрузка его контекста, выполнение и сохранение контекста — начинается с момента запуска системы и продолжается до ее выключения.

Таков вкратце механизм планирования работы множества потоков. Детали мы обсудим позже, но главное я уже показал. Все очень просто, да? Windows потому и называется системой с вытесняющей многозадачностью, что в любой момент может приостановить любой поток и вместо него запустить другой. Как Вы еще увидите, этим механизмом можно управлять, правда, крайне ограниченно. Всегда помните: Вы не в состоянии гарантировать, что Ваш поток будет выполняться непрерывно, что никакой другой поток не получит доступ к процессору и т. д.



Меня часто спрашивают, как сделать так, чтобы поток гарантированно запускался в течение определенного времени после какого-нибудь события — например, не позднее чем через миллисекунду после приема данных с последовательного порта? Ответ прост: никак. Такие требования можно предъявлять к операционным системам реального времени, но Windows к ним не относится. Лишь операционная система реального времени имеет полное представление о характеристиках аппаратных средств, на которых она работает (об интервалах запаздывания контроллеров жестких дисков, клавиатуры и т. д.). А создавая Windows, Microsoft ставила другую цель: обеспечить поддержку максимально широкого спектра оборудования — различных процессоров, дисковых устройств, сетей и др. Короче говоря, Windows не является операционной системой реального времени.

Хочу особо подчеркнуть, что система планирует выполнение только тех потоков, которые могут получать процессорное время, но большинство потоков в системе к таковым не относится. Так, у некоторых объектов-потоков значение счетчика простоеев (suspend count) больше 0, а значит, соответствующие потоки приостановлены и не получают процессорное время. Вы можете создать приостановленный поток вызовом *CreateProcess* или *CreateThread* с флагом *CREATE_SUSPENDED*. (В следующем разделе я расскажу и о таких функциях, как *SuspendThread* и *ResumeThread*.)

Кроме приостановленных, существуют и другие потоки, не участвующие в распределении процессорного времени, — они ожидают каких-либо событий. Например, если Вы запускаете Notepad и не работаете в нем с текстом, его поток бездействует, а система не выделяет процессорное время тем, кому нечего делать. Но стоит лишь сместить его окно, прокрутить в нем текст или что-то ввести, как система автоматически включит поток Notepad в число планируемых. Это вовсе не означает, что поток Notepad тут же начнет выполняться. Просто система учитывает его при планировании потоков и когда-нибудь выделит ему время — по возможности в ближайшем будущем.

Приостановка и возобновление потоков

В объекте ядра «поток» имеется переменная — счетчик числа простоев данного потока. При вызове *CreateProcess* или *CreateThread* он инициализируется значением, равным 1, которое запрещает системе выделять новому потоку процессорное время. Такая схема весьма разумна: сразу после создания поток не готов к выполнению, ему нужно время для инициализации.

После того как поток полностью инициализирован, *CreateProcess* или *CreateThread* проверяет, не передан ли ей флаг *CREATE_SUSPENDED*, и, если да, возвращает управление, оставив поток в приостановленном состоянии. В ином случае счетчик простоев обнуляется, и поток включается в число планируемых — если только он не ждет какого-то события (например, ввода с клавиатуры).

Создав поток в приостановленном состоянии, Вы можете настроить некоторые его свойства (например, приоритет, о котором мы поговорим позже). Закончив настройку, Вы должны разрешить выполнение потока. Для этого вызовите *ResumeThread* и передайте описатель потока, возвращенный функцией *CreateThread* (описатель можно взять и из структуры, на которую указывает параметр *ppiProcInfo*, передаваемый в *CreateProcess*).

```
DWORD ResumeThread(HANDLE hThread);
```

Если вызов *ResumeThread* прошел успешно, она возвращает предыдущее значение счетчика простоеев данного потока; в ином случае — 0xFFFFFFFF.

Выполнение отдельного потока можно приостанавливать несколько раз. Если поток приостановлен 3 раза, то и возобновлен он должен быть тоже 3 раза — лишь тогда система выделит ему процессорное время. Выполнение потока можно приостановить не только при его создании с флагом *CREATE_SUSPENDED*, но и вызовом *SuspendThread*:

```
DWORD SuspendThread(HANDLE hThread);
```

Любой поток может вызвать эту функцию и приостановить выполнение другого потока (конечно, если его описатель известен). Хоть об этом нигде и не говорится (но я все равно скажу!), приостановить свое выполнение поток способен сам, а возобновить себя без посторонней помощи — нет. Как и *ResumeThread*, функция *SuspendThread* возвращает предыдущее значение счетчика простоеев данного потока. Поток можно приостанавливать не более чем *MAXIMUM_SUSPEND_COUNT* раз (в файле WinNT.h это значение определено как 127). Обратите внимание, что *SuspendThread* в режиме ядра работает асинхронно, но в пользовательском режиме не выполняется, пока поток остается в приостановленном состоянии.

Создавая реальное приложение, будьте осторожны с вызовами *SuspendThread*, так как нельзя заранее сказать, чем будет заниматься его поток в момент приостановки. Например, он пытается выделить память из кучи и поэтому заблокировал к ней доступ. Тогда другим потокам, которым тоже нужна динамическая память, придется ждать его возобновления. *SuspendThread* безопасна только в том случае, когда Вы точно знаете, что делает (или может делать) поток, и предусматриваете все меры для исключения вероятных проблем и взаимной блокировки потоков. (О взаимной блокировке и других проблемах синхронизации потоков я расскажу в главах 8, 9 и 10.)

Приостановка и возобновление процессов

В Windows понятия «приостановка» и «возобновление» неприменимы к процессам, так как они не участвуют в распределении процессорного времени. Однако меня не раз спрашивали, как одним махом приостановить все потоки определенного процесса. Это можно сделать из другого процесса, причем он должен быть отладчиком и, в частности, вызывать функции вроде *WaitForDebugEvent* и *ContinueDebugEvent*.

Других способов приостановки всех потоков процесса в Windows нет: программа, выполняющая такую операцию, может «потерять» новые потоки. Система должна как-то приостанавливать в этот период не только все существующие, но и вновь создаваемые потоки. Microsoft предпочла встроить эту функциональность в системный механизм отладки.

Вам, конечно, не удастся написать идеальную функцию *SuspendProcess*, но вполне по силам добиться ее удовлетворительной работы во многих ситуациях. Вот мой вариант функции *SuspendProcess*.

```
VOID SuspendProcess(DWORD dwProcessID, BOOL fSuspend) {  
  
    // получаем список потоков в системе  
    HANDLE hSnapshot = CreateToolhelp32Snapshot(  
        TH32CS_SNAPTHREAD, dwProcessID);  
  
    if (hSnapshot != INVALID_HANDLE_VALUE) {  
  
        // просматриваем список потоков  
        THREADENTRY32 te = { sizeof(te) };  
        BOOL f0k = Thread32First(hSnapshot, &te);  
        for (; f0k; f0k = Thread32Next(hSnapshot, &te)) {  
  
            // относится ли данный поток к нужному процессу?  
            if (te.th32OwnerProcessID == dwProcessID) {  
  
                // пытаемся получить описатель потока по его идентификатору  
                HANDLE hThread = OpenThread(THREAD_SUSPEND_RESUME,  
                    FALSE, te.th32ThreadID);  
  
                if (hThread != NULL) {  
  
                    // приостанавливаем или возобновляем поток  
                    if (fSuspend)  
                        SuspendThread(hThread);  
                    else  
                        ResumeThread(hThread);  
                }  
                CloseHandle(hThread);  
            }  
        }  
        CloseHandle(hSnapshot);  
    }  
}
```

Для перечисления списка потоков я использую ToolHelp-функции (они рассматривались в главе 4). Определив потоки нужного процесса, я вызываю *OpenThread*:

```
HANDLE OpenThread(  
    DWORD dwDesiredAccess,  
    BOOL bInheritHandle,  
    DWORD dwThreadId);
```

Это новая функция, которая появилась в Windows 2000. Она находит объект ядра «поток» по идентификатору, указанному в *dwThreadId*, увеличивает его счетчик пользователей на 1 и возвращает описатель объекта. Получив описатель, я могу передать его в *SuspendThread* (или *ResumeThread*). *OpenThread* имеется только в Windows 2000, поэтому моя функция *SuspendProcess* не будет работать ни в Windows 95/98, ни в Windows NT 4.0.

Вероятно, Вы уже догадались, почему *SuspendProcess* будет срабатывать не во всех случаях: при перечислении могут создаваться новые и уничтожаться существующие потоки. После вызова *CreateToolhelp32Snapshot* в процессе может появиться новый поток, который моя функция уже не увидит, а значит, и не приостановит. Впоследствии, когда я попытаюсь возобновить потоки, вновь вызвав *SuspendProcess*, она во-

зобновит поток, который собственно и не приостанавливался. Но может быть еще хуже: при перечислении текущий поток уничтожается и создается новый с тем же идентификатором. Тогда моя функция приостановит неизвестно какой поток (и даже непонятно в каком процессе).

Конечно, все эти ситуации крайне маловероятны, и, если Вы точно представляете, что делает интересующий Вас процесс, никаких проблем не будет. В общем, используйте мою функцию на свой страх и риск.

Функция *Sleep*

Поток может сообщить системе не выделять ему процессорное время на определенный период, вызвав:

```
VOID Sleep(DWORD dwMilliseconds);
```

Эта функция приостанавливает поток на *dwMilliseconds* миллисекунд. Отметим несколько важных моментов, связанных с функцией *Sleep*.

- Вызывая *Sleep*, поток добровольно отказывается от остатка выделенного ему кванта времени.
- Система прекращает выделять потоку процессорное время на период, *примерно* равный заданному. Все верно: если Вы укажете остановить поток на 100 мс, приблизительно на столько он и «заснет», хотя не исключено, что его сон продлится на несколько секунд или даже минут больше. Вспомните, Windows не является системой реального времени. Ваш поток может возобновиться в данный момент, но это зависит от того, какая ситуация сложится в системе к тому времени.
- Вы можете вызвать *Sleep* и передать в *dwMilliseconds* значение *INFINITE*, вообще запретив планировать поток. Но это не очень практично — куда лучше корректно завершить поток, освободив его стек и объект ядра.
- Вы можете вызвать *Sleep* и передать в *dwMilliseconds* нулевое значение. Тогда Вы откажетесь от остатка своего кванта времени и заставите систему подключить к процессору другой поток. Однако система может снова запустить Ваш поток, если других планируемых потоков с тем же приоритетом нет.

Переключение потоков

Функция *SwitchToThread* позволяет подключить к процессору другой поток (если он есть):

```
BOOL SwitchToThread();
```

Когда Вы вызываете эту функцию, система проверяет, есть ли поток, которому не хватает процессорного времени. Если нет, *SwitchToThread* немедленно возвращает управление, а если да, планировщик отдает ему дополнительный квант времени (приоритет этого потока может быть ниже, чем у вызывающего). По истечении этого кванта планировщик возвращается в обычный режим работы.

SwitchToThread позволяет потоку, которому не хватает процессорного времени, отнять этот ресурс у потока с более низким приоритетом. Она возвращает FALSE, если на момент ее вызова в системе нет ни одного потока, готового к исполнению; в ином случае — ненулевое значение.

Вызов *SwitchToThread* аналогичен вызову *Sleep* с передачей в *dwMilliseconds* нулевого значения. Разница лишь в том, что *SwitchToThread* дает возможность выполнять потоки с более низким приоритетом, которым не хватает процессорного времени, а *Sleep* действует без оглядки на «голодающие» потоки.

WINDOWS 98 В Windows 98 функция *SwitchToThread* лишь определена, но не реализована.

Определение периодов выполнения потока

Иногда нужно знать, сколько времени затрачивает поток на выполнение той или иной операции. Многие в таких случаях пишут что-то вроде этого:

```
// получаем стартовое время  
DWORD dwStartTime = GetTickCount();  
  
// здесь выполняем какой-нибудь сложный алгоритм  
  
// вычитаем стартовое время из текущего  
DWORD dwElapsed = GetTickCount() - dwStartTime;
```

Этот код основан на простом допущении, что он не будет прерван. Но в операционной системе с вытесняющей многозадачностью никто не знает, когда поток получит процессорное время, и результат будет сильно искажен. Что нам здесь нужно, так это функция, которая сообщает время, затраченное процессором на обработку данного потока. К счастью, в Windows есть такая функция:

```
BOOL GetThreadTimes(  
    HANDLE hThread,  
    PFILETIME pftCreationTime,  
    PFILETIME pftExitTime,  
    PFILETIME pftKernelTime,  
    PFILETIME pftUserTime);
```

GetThreadTimes возвращает четыре временных параметра:

Показатель времени	Описание
Время создания (creation time)	Абсолютная величина, выраженная в интервалах по 100 нс. Отсчитывается с полуночи 1 января 1601 года по Гринвичу до момента создания потока.
Время завершения (exit time)	Абсолютная величина, выраженная в интервалах по 100 нс. Отсчитывается с полуночи 1 января 1601 года по Гринвичу до момента завершения потока. Если поток все еще выполняется, этот показатель имеет неопределенное значение.
Время выполнения ядра (kernel time)	Относительная величина, выраженная в интервалах по 100 нс. Сообщает время, затраченное этим потоком на выполнение кода операционной системы.
Время выполнения User (User time)	Относительная величина, выраженная в интервалах по 100 нс. Сообщает время, затраченное потоком на выполнение кода приложения.

С помощью этой функции можно определить время, необходимое для выполнения сложного алгоритма:

```
_int64 FileTimeToQuadWord(PFILETIME pft) {
    return(Int64ShlMod32(pft->dwHighDateTime, 32) | pft->dwLowDateTime);
}

void PerformLongOperation () {

    FILETIME ftKernelTimeStart, ftKernelTimeEnd;
    FILETIME ftUserTimeStart, ftUserTimeEnd;
    FILETIME ftDummy;
    _int64 qwKernelTimeElapsed, qwUserTimeElapsed, qwTotalTimeElapsed;

    // получаем начальные показатели времени
    GetThreadTimes(GetCurrentThread(), &ftDummy, &ftDummy,
                   &ftKernelTimeStart, &ftUserTimeStart);

    // здесь выполняем сложный алгоритм

    // получаем конечные показатели времени
    GetThreadTimes(GetCurrentThread(), &ftDummy, &ftDummy,
                   &ftKernelTimeEnd, &ftUserTimeEnd);

    // получаем значения времени, затраченного на выполнение ядра и User,
    // преобразуя начальные и конечные показатели времени из FILETIME
    // в четверенные слова, а затем вычитая начальные показатели из конечных
    qwKernelTimeElapsed = FileTimeToQuadWord(&ftKernelTimeEnd) -
                           FileTimeToQuadWord(&ftKernelTimeStart);

    qwUserTimeElapsed = FileTimeToQuadWord(&ftUserTimeEnd) -
                        FileTimeToQuadWord(&ftUserTimeStart);

    // получаем общее время, складывая время выполнения ядра и User
    qwTotalTimeElapsed = qwKernelTimeElapsed + qwUserTimeElapsed;

    // общее время хранится в qwTotalTimeElapsed
}
```

Заметим, что существует еще одна функция, аналогичная *GetThreadTimes* и применимая ко всем потокам в процессе:

```
BOOL GetProcessTimes(
    HANDLE hProcess,
    PFILETIME pftCreationTime,
    PFILETIME pftExitTime,
    PFILETIME pftKernelTime,
    PFILETIME pftUserTime);
```

GetProcessTimes возвращает временные параметры, суммированные по всем потокам (даже уже завершенным) в указанном процессе. Так, время выполнения ядра будет суммой периодов времени, затраченного всеми потоками процесса на выполнение кода операционной системы.

WINDOWS 98 К сожалению, в Windows 98 функции *GetThreadTimes* и *GetProcessTimes* определены, но не реализованы. Так что в Windows 98 нет надежного механизма, с помощью которого можно было бы определить, сколько процессорного времени выделяется потоку или процессу.

GetThreadTimes не годится для высокоточного измерения временных интервалов — для этого в Windows предусмотрено две специальные функции:

```
BOOL QueryPerformanceFrequency(LARGE_INTEGER* pliFrequency);  
BOOL QueryPerformanceCounter(LARGE_INTEGER* pliCount);
```

Они построены на том допущении, что поток не вытесняется, поскольку высокоточные измерения проводятся, как правило, в очень быстро выполняемых блоках кода. Чтобы слегка упростить работу с этими функциями, я создал следующий C++-класс:

```
class CStopwatch {  
public:  
    CStopwatch() { QueryPerformanceFrequency(&m_liPerfFreq); Start(); }  
  
    void Start() { QueryPerformanceCounter(&m_liPerfStart); }  
  
    __int64 Now() const { // возвращает число миллисекунд после вызова Start  
        LARGE_INTEGER liPerfNow;  
        QueryPerformanceCounter(&liPerfNow);  
        return(((liPerfNow.QuadPart - m_liPerfStart.QuadPart) * 1000)  
               / m_liPerfFreq.QuadPart);  
    }  
  
private:  
    LARGE_INTEGER m_liPerfFreq; // количество отсчетов в секунду  
    LARGE_INTEGER m_liPerfStart; // начальный отсчет  
};
```

Я применяю этот класс так:

```
// создаю секундомер (начинающий отсчет с текущего момента времени)  
CStopwatch stopwatch;  
  
// здесь я помещаю код, время выполнения которого нужно измерить  
  
// определяю, сколько времени прошло  
__int64 qwElapsedTime = stopwatch.Now();  
  
// qwElapsedTime сообщает длительность выполнения в миллисекундах
```

Структура CONTEXT

К этому моменту Вы должны понимать, какую важную роль играет структура CONTEXT в планировании потоков. Система сохраняет в ней состояние потока перед самым отключением его от процессора, благодаря чему его выполнение возобновляется с того места, где было прервано.

Вы, наверное, удивитесь, но в документации Platform SDK структуре CONTEXT отведен буквально один абзац:

«В структуре CONTEXT хранятся данные о состоянии регистров с учетом специфики конкретного процессора. Она используется системой для выполнения различных внутренних операций. В настоящее время такие структуры определены для процессоров Intel, MIPS, Alpha и PowerPC. Соответствующие определения см. в заголовочном файле WinNT.h.»

В документации нет ни слова об элементах этой структуры, набор которых зависит от типа процессора. Фактически CONTEXT — единственная из всех структур Windows, специфичная для конкретного процессора.

Так из чего же состоит структура CONTEXT? Давайте посмотрим. Ее элементы четко соответствуют регистрам процессора. Например, для процессоров *x86* в число элементов входят *Eax*, *Ebx*, *Ecx*, *Edx* и т. д., а для процессоров Alpha — *IntVO*, *IntT0*, *IntT1*, *IntSO*, *IntRa*, *IntZero* и др. Структура CONTEXT для процессоров *x86* выглядит так:

```
typedef struct _CONTEXT {
    // ...
    // Флаги, управляющие содержимым записи CONTEXT.
    //
    // Если запись контекста используется как входной параметр, тогда раздел,
    // управляемый флагом (когда он установлен), считается содержащим
    // действительные значения. Если запись контекста используется для
    // модификации контекста потока, то изменяются только те разделы, для
    // которых флаг установлен.
    //
    // Если запись контекста используется как входной и выходной параметр
    // для захвата контекста потока, возвращаются только те разделы контекста,
    // для которых установлены соответствующие флаги. Запись контекста никогда
    // не используется только как выходной параметр.
    //
    DWORD ContextFlags;

    //
    // Этот раздел определяется/возвращается, когда в ContextFlags установлен
    // флаг CONTEXT_DEBUG_REGISTERS. Заметьте, что CONTEXT_DEBUG_REGISTERS
    // не включаются в CONTEXT_FULL.
    //

    DWORD Dr0;
    DWORD Dr1;
    DWORD Dr2;
    DWORD Dr3;
    DWORD Dr6;
    DWORD Dr7;

    //
    // Этот раздел определяется/возвращается, когда в ContextFlags
    // установлен флаг CONTEXT_FLOATING_POINT.
    //

    FLOATING_SAVE_AREA FloatSave;
```

см. след. стр.

```
//  
// Этот раздел определяется/возвращается, когда в ContextFlags  
// установлен флаг CONTEXT_SEGMENTS.  
//  
DWORD SegGs;  
DWORD SegFs;  
DWORD SegEs;  
DWORD SegDs;  
  
//  
// Этот раздел определяется/возвращается, когда в ContextFlags  
// установлен флаг CONTEXT_INTEGER.  
//  
DWORD Edi;  
DWORD Esi;  
DWORD Ebx;  
DWORD Edx;  
DWORD ECX;  
DWORD Eax;  
  
//  
// Этот раздел определяется/возвращается, когда в ContextFlags  
// установлен флаг CONTEXT_CONTROL.  
//  
DWORD Ebp;  
DWORD Eip;  
DWORD SegCs;      // следует очистить  
DWORD EFlags;     // следует очистить  
DWORD Esp;  
DWORD SegSs;  
  
//  
// Этот раздел определяется/возвращается, когда в ContextFlags  
// установлен флаг CONTEXT_EXTENDED_REGISTERS.  
// Формат и смысл значений зависят от типа процессора.  
//  
BYTE ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];  
}  
} CONTEXT;
```

Эта структура разбита на несколько разделов. Раздел CONTEXT_CONTROL содержит управляющие регистры процессора: указатель команд, указатель стека, флаги и адрес возврата функции. (В отличие от x86, который при вызове функции помещает адрес возврата в стек, процессор Alpha сохраняет адрес возврата в одном из регистров.) Раздел CONTEXT_INTEGER соответствует целочисленным регистрам процессора, CONTEXT_FLOATING_POINT — регистрам с плавающей точкой, CONTEXT_SEGMENTS — сегментным регистрам (только для x86), CONTEXT_DEBUG_REGISTERS — регистрам, предназначенным для отладки (только для x86), а CONTEXT_EXTENDED_REGISTERS — дополнительным регистрам (только для x86).

Windows фактически позволяет заглянуть внутрь объекта ядра «поток» и получить сведения о текущем состоянии регистров процессора. Для этого предназначена функция:

```
BOOL GetThreadContext(
    HANDLE hThread,
    PCONTEXT pContext);
```

Создайте экземпляр структуры CONTEXT, инициализируйте нужные флаги (в элементе *ContextFlags*) и передайте функции *GetThreadContext* адрес этой структуры. Функция поместит значения в элементы, сведения о которых Вы запросили.

Прежде чем обращаться к *GetThreadContext*, приостановите поток вызовом *SuspendThread*, иначе поток может быть подключен к процессору, и значения регистров существенно изменятся. На самом деле у потока есть два контекста: пользовательского режима и режима ядра. *GetThreadContext* возвращает лишь первый из них. Если Вы вызываете *SuspendThread*, когда поток выполняет код операционной системы, пользовательский контекст можно считать достоверным, даже несмотря на то что поток еще не остановлен (он все равно не выполнит ни одной команды пользовательского кода до последующего возобновления).

Единственный элемент структуры CONTEXT, которому не соответствует какой-либо регистр процессора, — *ContextFlags*. Присутствуя во всех вариантах этой структуры независимо от типа процессора, он подсказывает функции *GetThreadContext*, значения каких регистров Вы хотите узнать. Например, чтобы получить значения управляющих регистров для потока, напишите что-то вроде:

```
// создаем экземпляр структуры CONTEXT
CONTEXT Context;

// сообщаем системе, что нас интересуют сведения
// только об управляющих регистрах
Context.ContextFlags = CONTEXT_CONTROL;

// требуем от системы информацию о состоянии
// регистров процессора для данного потока
GetThreadContext(hThread, &Context);

// действительные значения содержат элементы структуры CONTEXT,
// соответствующие управляющим регистрам, остальные значения
// не определены
```

Перед вызовом *GetThreadContext* надо инициализировать элемент *ContextFlags*. Чтобы получить значения как управляющих, так и целочисленных регистров, инициализируйте его так:

```
// сообщаем системе, что нас интересуют
// управляющие и целочисленные регистры
Context.ContextFlags = CONTEXT_CONTROL | CONTEXT_INTEGER;
```

Есть еще один идентификатор, позволяющий узнать значения важнейших регистров (т. е. используемых, по мнению Microsoft, чаще всего):

```
// сообщаем системе, что нас интересуют
// все значимые регистры
Context.ContextFlags = CONTEXT_FULL;
```

CONTEXT_FULL определен в файле WinNT.h, как показано в таблице.

Тип процессора	Определение CONTEXT_FULL
x86	CONTEXT_CONTROL CONTEXT_INTEGER CONTEXT_SEGMENTS
Alpha	CONTEXT_CONTROL CONTEXT_FLOATING_POINT CONTEXT_INTEGER

После возврата из *GetThreadContext* Вы легко проверите значения любых регистров для потока, но помните, что такой код зависит от типа процессора. В следующей таблице перечислены элементы структуры CONTEXT, соответствующие указателям команд и стека для разных типов процессоров.

Тип процессора	Указатель команд	Указатель стека
x86	CONTEXT.Eip	CONTEXT.Esp
Alpha	CONTEXT.Fir	CONTEXT.IntSp

Даже удивительно, какой мощный инструмент дает Windows в руки разработчику! Но есть вещь, от которой Вы придете в полный восторг: значения элементов CONTEXT можно изменять и передавать объекту ядра «поток» с помощью функции *SetThreadContext*.

```
BOOL SetThreadContext(  
    HANDLE hThread,  
    CONST CONTEXT *pContext);
```

Перед этой операцией поток тоже нужно приостановить, иначе результаты могут быть непредсказуемыми.

Прежде чем обращаться к *SetThreadContext*, инициализируйте элемент *ContextFlags*, как показано ниже.

```
CONTEXT Context;
```

```
// приостанавливаем поток  
SuspendThread(hThread);  
  
// получаем регистры для контекста потока  
Context.ContextFlags = CONTEXT_CONTROL;  
GetThreadContext(hThread, &Context);  
  
// устанавливаем указатель команд по своему выбору;  
// в нашем примере присваиваем значение 0x00010000  
#if defined(_ALPHA_)  
Context.Fir = 0x00010000;  
#elif defined(_X86_)  
Context.Eip = 0x00010000;  
#else  
#error Module contains CPU-specific code; modify and recompile.  
#endif  
  
// вносим изменения в регистры потока; ContextFlags  
// можно и не инициализировать, так как это уже сделано  
Context.ControlFlags = CONTEXT_CONTROL;  
SetThreadContext(hThread, &Context);  
  
// возобновляем выполнение потока; оно начнется с адреса 0x00010000  
ResumeThread(hThread);
```

Этот код, вероятно, приведет к ошибке защиты (нарушению доступа) в удаленном потоке; система сообщит о необработанном исключении, и удаленный процесс будет закрыт. Все верно — не Ваш, а удаленный. Вы благополучно обрушили другой процесс, оставив свой в целости и сохранности!

Функции *GetThreadContext* и *SetThreadContext* наделяют Вас огромной властью над потоками, но пользоваться ею нужно с осторожностью. Вызывают их лишь считанные приложения. Эти функции предназначены для отладчиков и других инструментальных средств, хотя обращаться к ним можно из любых программ.

Подробнее о структуре CONTEXT мы поговорим в главе 24.

Приоритеты потоков

В начале главы я сказал, что поток получает доступ к процессору на 20 мс, после чего планировщик переключает процессор на выполнение другого потока. Так происходит, только если у всех потоков один приоритет, но на самом деле в системе существуют потоки с разными приоритетами, а это меняет порядок распределения процессорного времени.

Каждому потоку присваивается уровень приоритета — от 0 (самый низкий) до 31 (самый высокий). Решая, какому потоку выделить процессорное время, система сначала рассматривает только потоки с приоритетом 31 и подключает их к процессору по принципу карусели. Если поток с приоритетом 31 не исключен из планирования, он немедленно получает квант времени, по истечении которого система проверяет, есть ли еще один такой поток. Если да, он тоже получает свой квант процессорного времени.

Пока в системе имеются планируемые потоки с приоритетом 31, ни один поток с более низким приоритетом процессорного времени не получает. Такая ситуация называется «голоданием» (starvation). Она наблюдается, когда потоки с более высоким приоритетом так интенсивно используют процессорное время, что остальные практически не работают. Вероятность этой ситуации намного ниже в многопроцессорных системах, где потоки с приоритетами 31 и 30 могут выполняться одновременно. Система всегда старается, чтобы процессоры были загружены работой, и они пропстаивают только в отсутствие планируемых потоков.

На первый взгляд, в системе, организованной таким образом, у потоков с низким приоритетом нет ни единого шанса на исполнение. Но, как я уже говорил, зачастую потоки как раз и не нужно выполнять. Например, если первичный поток Вашего процесса вызывает *GetMessage*, а система видит, что никаких сообщений пока нет, она приостанавливает его выполнение, отнимает остаток неиспользованного времени и тут же подключает к процессору другой ожидающий поток. И пока в системе не появятся сообщения для потока Вашего процесса, он будет пропстаивать — система не станет тратить на него процессорное время. Но вот в очереди этого потока появляется сообщение, и система сразу же подключает его к процессору (если только в этот момент не выполняется поток с более высоким приоритетом).

А теперь обратите внимание на еще один момент. Потоки с более высоким приоритетом всегда вытесняют потоки с более низким приоритетом независимо от того, исполняются последние или нет. Допустим, процессор исполняет поток с приоритетом 5, и тут система обнаруживает, что поток с более высоким приоритетом готов к выполнению. Что будет? Система остановит поток с более низким приоритетом — даже если не истек отведенный ему квант процессорного времени — и подключит к процессору поток с более высоким приоритетом (и, между прочим, выдаст ему полный квант времени).

Кстати, при загрузке системы создается особый поток — *поток обнуления страниц* (zero page thread), которому присваивается нулевой уровень приоритета. Ни один поток, кроме этого, не может иметь нулевой уровень приоритета. Он обнуляет свободные страницы в оперативной памяти при отсутствии других потоков, требующих внимания со стороны системы.

Абстрагирование приоритетов

Создавая планировщик потоков, разработчики из Microsoft прекрасно понимали, что он не подойдет на все случаи жизни. Они также осознавали, что со временем «назначение» компьютера может измениться. Например, в момент выпуска Windows NT создание приложений с поддержкой OLE еще только начиналось. Теперь такие приложения — обычное дело. Кроме того, значительно расширилось применение игрового программного обеспечения, ну и, конечно же, Интернета.

Алгоритм планирования потоков существенно влияет на выполнение приложений. С самого начала разработчики Microsoft понимали, что его придется изменять по мере того, как будут расширяться сферы применения компьютеров. Microsoft гарантирует, что наши программы будут работать и в следующих версиях Windows. Как же ей удается изменять внутреннее устройство системы, не нарушая работоспособность наших программ? Ответ в том, что:

- планировщик документируется не полностью;
- Microsoft не разрешает в полной мере использовать все особенности планировщика;
- Microsoft предупреждает, что алгоритм работы планировщика постоянно меняется, и не рекомендует писать программы в расчете на текущий алгоритм.

Windows API предоставляет слой абстрагирования от конкретного алгоритма работы планировщика, запрещая прямое обращение к планировщику. Вместо этого Вы вызываете функции Windows, которые «интерпретируют» Ваши параметры в зависимости от версии системы. Я буду рассказывать именно об этом слое абстрагирования.

Проектируя свое приложение, Вы должны учитывать возможность параллельного выполнения других программ. Следовательно, Вы обязаны выбирать класс приоритета, исходя из того, насколько «отзывчивой» должна быть Ваша программа. Согласен, такая формулировка довольно туманна, но так и задумано: Microsoft не желает обеспечить ничего такого, что могло бы нарушить работу Вашего кода в будущем.

Windows поддерживает шесть классов приоритета: idle (простаивающий), below normal (ниже обычного), normal (обычный), above normal (выше обычного), high (высокий) и realtime (реального времени). Самый распространенный класс приоритета, естественно, — normal; его использует 99% приложений. Классы приоритета показаны в следующей таблице.

Класс приоритета	Описание
Real-time	Потоки в этом процессе обязаны немедленно реагировать на события, обеспечивая выполнение критических по времени задач. Такие потоки вытесняют даже компоненты операционной системы. Будьте крайне осторожны с этим классом.
High	Потоки в этом процессе тоже должны немедленно реагировать на события, обеспечивая выполнение критических по времени задач. Этот класс присвоен, например, Task Manager, что дает возможность пользователю закрывать больше неконтролируемые процессы.

продолжение

Класс приоритета	Описание
Above normal	Класс приоритета, промежуточный между normal и high. Это новый класс, введенный в Windows 2000.
Normal	Потоки в этом процессе не предъявляют особых требований к выделению им процессорного времени.
Below normal	Класс приоритета, промежуточный между normal и idle. Это новый класс, введенный в Windows 2000.
Idle	Потоки в этом процессе выполняются, когда система не занята другой работой. Этот класс приоритета обычно используется для утилит, работающих в фоновом режиме, экранных заставок и приложений, собирающих статистическую информацию.

Приоритет idle идеален для программ, выполняемых, только когда системе больше нечего делать. Примеры таких программ — экранные заставки и средства мониторинга. Компьютер, не используемый в интерактивном режиме, может быть занят другими задачами (действуя, скажем, в качестве файлового сервера), и их потокам незачем конкурировать с экранной заставкой за доступ к процессору. Средства мониторинга, собирающие статистическую информацию о системе, тоже не должны мешать выполнению более важных задач.

Класс приоритета high следует использовать лишь при крайней необходимости. Может, Вы этого и не знаете, но Explorer выполняется с высоким приоритетом. Большинство часть времени его потоки простоявают, готовые пробудиться, как только пользователь нажмет какую-нибудь клавишу или щелкнет кнопку мыши. Пока потоки Explorer простоявают, система не выделяет им процессорное время, что позволяет выполнять потоки с более низким приоритетом. Но вот пользователь нажал, скажем, Ctrl+Esc, и система пробуждает поток Explorer. (Комбинация клавиш Ctrl+Esc попутно открывает меню Start.) Если в данный момент исполняются потоки с более низким приоритетом, они немедленно вытесняются, и начинает работать поток Explorer. Microsoft разработала Explorer именно так потому, что любой пользователь — независимо от текущей ситуации в системе — ожидает мгновенной реакции оболочки на свои команды. В сущности, окна Explorer можно открывать, даже когда все потоки с более низким приоритетом зависают в бесконечных циклах. Обладая более высоким приоритетом, потоки Explorer вытесняют поток, исполняющий бесконечный цикл, и дают возможность закрыть зависший процесс.

Надо отметить высокую степень продуманности Explorer. Основную часть времени он просто «спит», не требуя процессорного времени. Будь это не так, вся система работала бы гораздо медленнее, а многие приложения просто не отзывались бы на действия пользователя.

Классом приоритета real-time почти никогда не стоит пользоваться. На самом деле в ранних бета-версиях Windows NT 3.1 присвоение этого класса приоритета приложениям даже не предусматривалось, хотя операционная система поддерживала эту возможность. Real-time — чрезвычайно высокий приоритет, и, поскольку большинство потоков в системе (включая управляющие самой системой) имеет более низкий приоритет, процесс с таким классом окажет на них сильное влияние. Так, потоки реального времени могут заблокировать необходимые операции дискового и сетевого ввода-вывода и привести к несвоевременной обработке ввода от мыши и клавиатуры — пользователь может подумать, что система зависла. У Вас должна быть очень веская причина для применения класса real-time — например, программе требуется

реагировать на события в аппаратных средствах с минимальной задержкой или выполнять быстротечную операцию, которую нельзя прерывать ни при каких обстоятельствах.



Процесс с классом приоритета real-time нельзя запустить, если пользователь не имеет привилегии Increase Scheduling Priority. По умолчанию такой привилегией обладает администратор и пользователь с расширенными полномочиями.

Конечно, большинство процессов имеет обычный класс приоритета. В Windows 2000 появилось два новых промежуточных класса — below normal и above normal. Microsoft добавила их, поскольку некоторые компании жаловались, что существующий набор классов приоритетов не дает нужной гибкости.

Выбрав класс приоритета, забудьте о том, как Ваша программа будет выполняться совместно с другими приложениями, и сосредоточьтесь на ее потоках. Windows поддерживает семь относительных приоритетов потоков: idle (простаивающий), lowest (нижний), below normal (ниже обычного), normal (обычный), above normal (выше обычного), highest (высший) и time-critical (критичный по времени). Эти приоритеты относительны классу приоритета процесса. Как обычно, большинство потоков использует обычный приоритет. Относительные приоритеты потоков описаны в следующей таблице.

Относительный приоритет потока	Описание
Time-critical	Поток выполняется с приоритетом 31 в классе real-time и с приоритетом 15 в других классах
Highest	Поток выполняется с приоритетом на два уровня выше обычного для данного класса
Above normal	Поток выполняется с приоритетом на один уровень выше обычного для данного класса
Normal	Поток выполняется с обычным приоритетом процесса для данного класса
Below normal	Поток выполняется с приоритетом на один уровень ниже обычного для данного класса
Lowest	Поток выполняется с приоритетом на два уровня ниже обычного для данного класса
Idle	Поток выполняется с приоритетом 16 в классе real-time и с приоритетом 1 в других классах

Итак, Вы присваиваете процессу некий класс приоритета и можете изменять относительные приоритеты потоков в пределах процесса. Заметьте, что я не сказал ни слова об уровнях приоритетов 0–31. Разработчики приложений не имеют с ними дела. Уровень приоритета формируется самой системой, исходя из класса приоритета процесса и относительного приоритета потока. А механизм его формирования — как раз то, чем Microsoft не хочет себя ограничивать. И действительно, этот механизм меняется практически в каждой версии системы.

В следующей таблице показано, как формируется уровень приоритета в Windows 2000, но не забывайте, что в Windows NT и тем более в Windows 95/98 этот механизм действует несколько иначе. Учтите также, что в будущих версиях Windows он вновь изменится.

Например, обычный поток в обычном процессе получает уровень приоритета 8. Поскольку большинство процессов имеет класс normal, а большинство потоков —

относительный приоритет *normal*, у основной части потоков в системе уровень приоритета равен 8.

Обычный поток в процессе с классом приоритета *high* получает уровень приоритета 13. Изменив класс приоритета процесса на *idle*, Вы снизите уровень приоритета того же потока до 4. Вспомните, что приоритет потока всегда относителен классу приоритета его процесса. Изменение класса приоритета процесса не влияет на относительные приоритеты его потоков, но сказывается на уровне их приоритета.

Относительный приоритет потока	Idle	Класс приоритета процесса				
		Below normal	Normal	Above normal	High	Real-time
Time-critical (критичный по времени)	15	15	15	15	15	31
Highest (высший)	6	8	10	12	15	26
Above normal (выше обычного)	5	7	9	11	14	25
Normal (обычный)	4	6	8	10	13	24
Below normal (ниже обычного)	3	5	7	9	12	23
Lowest (низший)	2	4	6	8	11	22
Idle (простаивающий)	1	1	1	1	1	16

Обратите внимание, что в таблице не показано, как задать уровень приоритета 0. Это связано с тем, что нулевой приоритет зарезервирован для потока обнуления страниц, и никакой другой поток не может иметь такой приоритет. Кроме того, уровни 17–21 и 27–30 в обычном приложении тоже недоступны. Вы можете пользоваться ими, только если пишете драйвер устройства, работающий в режиме ядра. И еще одно: уровень приоритета потока в процессе с классом *real-time* не может опускаться ниже 16, а потока в процессе с любым другим классом — подниматься выше 15.



Концепция класса приоритета вводит некоторых в заблуждение. Они делают отсюда вывод, будто процессы участвуют в распределении процессорного времени. Так вот, процессы никогда не получают процессорное время — оно выделяется лишь потокам. Класс приоритета процесса — сугубо абстрактная концепция, введенная Microsoft с единственной целью: скрыть от разработчика внутреннее устройство планировщика.



В общем случае поток с высоким уровнем приоритета должен быть активен как можно меньше времени. При появлении у него какой-либо работы он тут же получает процессорное время. Выполнив минимальное количество команд, он должен снова вернуться в ждущий режим. С другой стороны, поток с низким уровнем приоритета может оставаться активным и занимать процессор довольно долго. Следуя этим правилам, Вы сохранитеющую отзывчивость операционной системы на действия пользователя.

Программирование приоритетов

Так как же процесс получает класс приоритета? Очень просто. Вызывая *CreateProcess*, Вы можете указать в ее параметре *fdwCreate* нужный класс приоритета. Идентификаторы этих классов приведены в следующей таблице.

Класс приоритета	Идентификатор
Real-time	REALTIME_PRIORITY_CLASS
High	HIGH_PRIORITY_CLASS
Above normal	ABOVE_NORMAL_PRIORITY_CLASS
Normal	NORMAL_PRIORITY_CLASS
Below normal	BELLOW_NORMAL_PRIORITY_CLASS
Idle	IDLE_PRIORITY_CLASS

Вам может показаться странным, что, создавая дочерний процесс, родительский сам устанавливает ему класс приоритета. За примером далеко ходить не надо — возьмем все тот же Explorer. При запуске из него какого-нибудь приложения новый процесс создается с обычным приоритетом. Но Explorer ведь не знает, что делает этот процесс и как часто его потокам надо выделять процессорное время. Поэтому в системе предусмотрена возможность изменения класса приоритета самим выполняемым процессом — вызовом функции *SetPriorityClass*:

```
BOOL SetPriorityClass(  
    HANDLE hProcess,  
    DWORD fdwPriority);
```

Эта функция меняет класс приоритета процесса, определяемого описателем *hProcess*, в соответствии со значением параметра *fdwPriority*. Последний должен содержать одно из значений, указанных в таблице выше. Поскольку *SetPriorityClass* принимает описатель процесса, Вы можете изменить приоритет любого процесса, выполняемого в системе, — если его описатель известен и у Вас есть соответствующие права доступа.

Обычно процесс пытается изменить свой класс приоритета. Вот как процесс может сам себе установить класс приоритета idle:

```
BOOL SetPriorityClass(GetCurrentProcess(), IDLE_PRIORITY_CLASS);
```

Парная ей функция *GetPriorityClass* позволяет узнать класс приоритета любого процесса:

```
DWORD GetPriorityClass(HANDLE hProcess);
```

Она возвращает, как Вы догадываетесь, один из ранее перечисленных флагов.

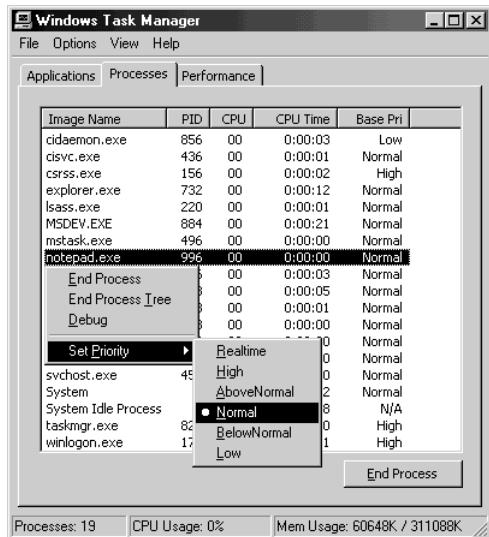
При запуске из оболочки командного процессора начальный приоритет программы тоже обычный. Однако, запуская ее командой Start, можно указать ключ, определяющий начальный приоритет. Так, следующая команда, введенная в оболочке командного процессора, заставит систему запустить приложение Calculator и присвоить ему приоритет idle:

```
C:\>START /LOW CALC.EXE
```

Команда Start допускает также ключи /BELOWNORMAL, /NORMAL, /ABOVENORMAL, /HIGH и /REALTIME, позволяющие начать выполнение программы с соответствующим классом приоритета. Разумеется, после запуска программа может вызвать *SetPriorityClass* и установить себе другой класс приоритета.

WINDOWS 98 В Windows 98 команда Start не поддерживает ни один из этих ключей. Из оболочки командного процессора Windows 98 процессы всегда запускаются с классом приоритета normal.

Task Manager в Windows 2000 дает возможность изменять класс приоритета процесса. На рисунке ниже показана вкладка Processes в окне Task Manager со списком выполняемых на данный момент процессов. В колонке Base Pri сообщается класс приоритета каждого процесса. Вы можете изменить его, выбрав процесс и указав другой класс в подменю Set Priority контекстного меню.



Только что созданный поток получает относительный приоритет `normal`. Почему `CreateThread` не позволяет задать относительный приоритет — для меня так и остается загадкой. Такая операция осуществляется вызовом функции:

```
BOOL SetThreadPriority(  
    HANDLE hThread,  
    int nPriority);
```

Разумеется, параметр *bThread* указывает на поток, чей приоритет Вы хотите изменить, а через *nPriority* передается один из идентификаторов (см. таблицу ниже).

Относительный приоритет потока	Идентификатор
Time-critical	THREAD_PRIORITY_TIME_CRITICAL
Highest	THREAD_PRIORITY_HIGHEST
Above normal	THREAD_PRIORITY_ABOVE_NORMAL
Normal	THREAD_PRIORITY_NORMAL
Below normal	THREAD_PRIORITY_BELOW_NORMAL
Lowest	THREAD_PRIORITY_LOWEST
Idle	THREAD_PRIORITY_IDLE

Функция *GetThreadPriority*, парная *SetThreadPriority*, позволяет узнать относительный приоритет потока:

```
int GetThreadPriority(HANDLE hThread);
```

Она возвращает один из идентификаторов, показанных в таблице выше.

Чтобы создать поток с относительным приоритетом idle, сделайте, например, так:

```
DWORD dwThreadID;
HANDLE hThread = CreateThread(NULL, 0, ThreadFunc, NULL,
    CREATE_SUSPENDED, &dwThreadID);
SetThreadPriority(hThread, THREAD_PRIORITY_IDLE);
ResumeThread(hThread);
CloseHandle(hThread);
```

Заметьте, что *CreateThread* всегда создает поток с относительным приоритетом *normal*. Чтобы присвоить потоку относительный приоритет *idle*, создайте приостановленный поток, передав в *CreateThread* флаг *CREATE_SUSPENDED*, а потом вызовите *SetThreadPriority* и установите нужный приоритет. Далее можно вызвать *ResumeThread*, и поток будет включен в число планируемых. Сказать заранее, когда поток получит процессорное время, нельзя, но планировщик уже учитывает его новый приоритет. Выполнив эти операции, Вы можете закрыть описатель потока, чтобы соответствующий объект ядра был уничтожен по завершении данного потока.



Ни одна Windows-функция не возвращает уровень приоритета потока. Такая ситуация создана преднамеренно. Вспомните, что Microsoft может в любой момент изменить алгоритм распределения процессорного времени. Поэтому при разработке приложений не стоит опираться на какие-то нюансы этого алгоритма. Используйте классы приоритетов процессов и относительные приоритеты потоков, и Ваши приложения будут нормально работать как в нынешних, так и в следующих версиях Windows.

Динамическое изменение уровня приоритета потока

Уровень приоритета, получаемый комбинацией относительного приоритета потока и класса приоритета процесса, которому принадлежит данный поток, называют *базовым уровнем приоритета потока*. Иногда система изменяет уровень приоритета потока. Обычно это происходит в ответ на некоторые события, связанные с вводом-выводом (например, на появление оконных сообщений или чтение с диска).

Так, поток с относительным приоритетом *normal*, выполняемый в процессе с классом приоритета *high*, имеет базовый приоритет 13. Если пользователь нажимает какую-нибудь клавишу, система помещает в очередь потока сообщение *WM_KEYDOWN*. А поскольку в очереди потока появилось сообщение, поток становится планируемым. При этом драйвер клавиатуры может заставить систему временно поднять уровень приоритета потока с 13 до 15 (действительное значение может отличаться в ту или другую сторону).

Процессор исполняет поток в течение отведенного отрезка времени, а по его истечении система снижает приоритет потока на 1, до уровня 14. Далее потоку вновь выделяется квант процессорного времени, по окончании которого система опять снижает уровень приоритета потока на 1. И теперь приоритет потока снова соответствует его базовому уровню.

Текущий уровень приоритета не может быть ниже базового. Кроме того, драйвер устройства, «разбудивший» поток, сам устанавливает величину повышения приоритета. И опять же Microsoft не документирует, насколько повышаются эти значения конкретными драйверами. Таким образом, она получает возможность тонко настраивать динамическое изменение приоритетов потоков в операционной системе, чтобы та максимально быстро реагировала на действия пользователя.

Система повышает приоритет только тех потоков, базовый уровень которых находится в пределах 1–15. Именно поэтому данный диапазон называется «областью динамического приоритета» (dynamic priority range). Система не допускает динамического повышения приоритета потока до уровней реального времени (более 15). Поскольку потоки с такими уровнями обслуживают системные функции, это ограничение не дает приложению нарушить работу операционной системы. И, кстати, система никогда не меняет приоритет потоков с уровнями реального времени (от 16 до 31).

Некоторые разработчики жаловались, что динамическое изменение приоритета системой отрицательно сказывается на производительности их приложений, и поэтому Microsoft добавила две функции, позволяющие отключать этот механизм:

```
BOOL SetProcessPriorityBoost(  
    HANDLE hProcess,  
    BOOL DisablePriorityBoost);  
  
BOOL SetThreadPriorityBoost(  
    HANDLE hThread,  
    BOOL DisablePriorityBoost);
```

SetProcessPriorityBoost заставляет систему включить или отключить изменение приоритетов всех потоков в указанном процессе, а *SetThreadPriorityBoost* действует применительно к отдельным потокам. Эти функции имеют свои аналоги, позволяющие определять, разрешено или запрещено изменение приоритетов:

```
BOOL GetProcessPriorityBoost(  
    HANDLE hProcess,  
    PBOOL pDisablePriorityBoost);  
  
BOOL GetThreadPriorityBoost(  
    HANDLE hThread,  
    PBOOL pDisablePriorityBoost);
```

Каждой из этих двух функций Вы передаете описатель нужного процесса или потока и адрес переменной типа *BOOL*, в которой и возвращается результат.

WINDOWS 98 В Windows 98 эти четыре функции определены, но не реализованы, и при вызове любой из них возвращается FALSE. Последующий вызов *GetLastError* дает *ERROR_CALL_NOT_IMPLEMENTED*.

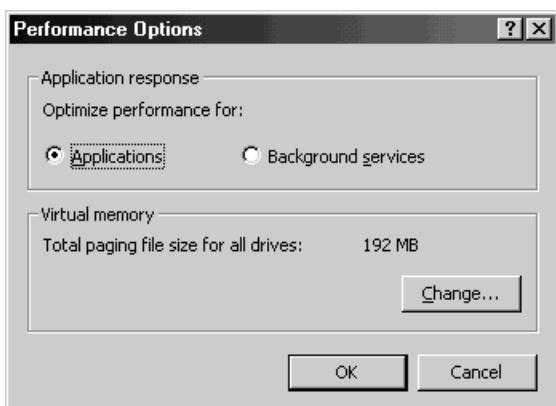
Есть еще одна ситуация, в которой система динамически повышает приоритет потока. Представьте, что поток с приоритетом 4 готов к выполнению, но не может получить доступ к процессору из-за того, что его постоянно занимают потоки с приоритетом 8. Это типичный случай «голодания» потока с более низким приоритетом. Обнаружив такой поток, не выполняемый на протяжении уже трех или четырех секунд, система поднимает его приоритет до 15 и выделяет ему двойную порцию времени. По его истечении потоку немедленно возвращается его базовый приоритет.

Подстройка планировщика для активного процесса

Когда пользователь работает с окнами какого-то процесса, последний считается *активным* (foreground process), а остальные процессы — *фоновыми* (background processes). Естественно, пользователь заинтересован в повышенной отзывчивости активного процесса по сравнению с фоновыми. Для этого Windows подстраивает алгоритм планирования потоков активного процесса. В Windows 2000, когда процесс становит-

ся активным, система выделяет его потокам более длительные кванты времени. Такая регулировка применяется только к процессам с классом приоритета normal.

Windows 2000 позволяет модифицировать работу этого механизма подстройки. Щелкнув кнопку Performance Options на вкладке Advanced диалогового окна System Properties, Вы открываете следующее окно.



Переключатель Applications включает подстройку планировщика для активного процесса, а переключатель Background Services — выключает (в этом случае оптимизируется выполнение фоновых сервисов). В Windows 2000 Professional по умолчанию выбирается переключатель Applications, а в остальных версиях Windows 2000 — переключатель Background Services, так как серверы редко используются в интерактивном режиме.

Windows 98 тоже позволяет подстраивать распределение процессорного времени для потоков активного процесса с классом приоритета normal. Когда процесс этого класса становится активным, система повышает на 1 приоритет его потоков, если их исходные приоритеты были lowest, below normal, normal, above normal или highest; приоритет потоков idle или time-critical не меняется. Поэтому поток с относительным приоритетом normal в активном процессе с классом приоритета normal имеет уровень приоритета 9, а не 8. Когда процесс вновь становится фоновым, приоритеты его потоков автоматически возвращаются к исходным уровням.

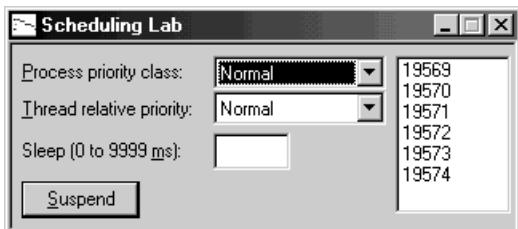
WINDOWS 98 Windows 98 не предусматривает возможности настройки этого механизма, так как не рассчитана на работу в качестве выделенного сервера.

Причина для таких изменений активных процессов очень проста: система дает им возможность быстрее реагировать на пользовательский ввод. Если бы приоритеты их потоков не менялись, то и обычный процесс фоновой печати, и обычный, но активный процесс, принимающий пользовательский ввод, — оба одинаково конкурировали бы за процессорное время. И тогда пользователь, набирая текст в активном приложении, заметил бы, что текст появляется на экране какими-то рывками. Но благодаря тому, что система повышает уровни приоритета потоков активного процесса, они получают преимущество над потоками обычных фоновых процессов.

Программа-пример Scheduling Lab

Эта программа, «07 SchedLab.exe» (см. листинг на рис. 7-1), позволяет экспериментировать с классами приоритетов процессов и относительными приоритетами потоков

и исследовать их влияние на общую производительность системы. Файлы исходного кода и ресурсов этой программы находятся в каталоге 07-SchedLab на компакт-диске, прилагаемом к книге. После запуска SchedLab открывается окно, показанное ниже.



Изначально первичный поток работает очень активно, и степень использования процессора подскакивает до 100%. Все, чем он занимается, — постоянно увеличивает исходное значение на 1 и выводит текущее значение в крайнее справа окно списка. Все эти числа не несут никакой смысловой информации; их появление просто демонстрирует, что поток чем-то занят. Чтобы прочувствовать, как повлияет на него изменение приоритета, запустите по крайней мере два экземпляра программы. Можете также открыть Task Manager и понаблюдать за нагрузкой на процессор, создаваемой каждым экземпляром.

В начале теста процессор будет загружен на 100%, и Вы увидите, что все экземпляры SchedLab получают примерно равные кванты процессорного времени. (Task Manager должен показать практически одинаковые процентные доли для всех ее экземпляров.) Как только Вы поднимете класс приоритета одного из экземпляров до above normal или high, львиную долю процессорного времени начнет получать именно этот экземпляр, а аналогичные показатели для других экземпляров резко упадут. Однако они никогда не опустятся до нуля — это действует механизм динамического повышения приоритета «голодающих» процессов. Теперь Вы можете самостоятельно поиграть с изменением классов приоритетов процессов и относительных приоритетов потоков. Возможность установки класса приоритета real-time я исключил намеренно, чтобы не нарушить работу операционной системы. Если Вы все же хотите поэкспериментировать с этим приоритетом, Вам придется модифицировать исходный текст моей программы.

Используя поле Sleep, можно приостановить первичный поток на заданное число миллисекунд в диапазоне от 0 до 9999. Попробуйте приостанавливать его хотя бы на 1 мс и посмотрите, сколько процессорного времени это позволит сэкономить. На своем ноутбуке с процессором Pentium II 300 МГц, я выиграл аж 99% — впечатляет!

Кнопка Suspend заставляет первичный поток создать дочерний поток, который приостанавливает родительский и выводит следующее окно.



Пока это окно открыто, первичный поток полностью отключается от процессора, а дочерний тоже не требует процессорного времени, так как ждет от пользователя дальнейших действий. Вы можете свободно перемещать это окно в пределах экрана или убрать его в сторону от основного окна программы. Поскольку первичный поток остановлен, основное окно не принимает оконных сообщений (в том числе

WM_PAINT). Это еще раз доказывает, что поток задержан. Закрыв окно с сообщением, Вы возобновите первичный поток, и нагрузка на процессор снова возрастет до 100%.

А теперь проведите еще один эксперимент. Откройте диалоговое окно Performance Options (я говорил о нем в предыдущем разделе) и выберите переключатель Background Services (или, наоборот, Application). Потом запустите несколько экземпляров моей программы с классом приоритета normal и выберите один из них, сделав его активным процессом. Вы сможете наглядно убедиться, как эти переключатели влияют на активные и фоновые процессы.



SchedLab.cpp

```

/*
Модуль: SchedLab.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*/

#include "..\CmnHdr.h"           /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include <process.h>             // для доступа к _beginthreadex
#include "Resource.h"

///////////

DWORD WINAPI ThreadFunc(PVOID pvParam) {
    HANDLE hThreadPrimary = (HANDLE) pvParam;
    SuspendThread(hThreadPrimary);
    chMB(
        "The Primary thread is suspended.\n"
        "It no longer responds to input and produces no output.\n"
        "Press OK to resume the primary thread & exit this secondary thread.\n");
    ResumeThread(hThreadPrimary);
    CloseHandle(hThreadPrimary);

    // во избежание взаимной блокировки после ResumeThread вызываем EnableWindow
    EnableWindow(
        GetDlgItem(FindWindow(NULL, TEXT("Scheduling Lab")), IDC_SUSPEND), TRUE);
    return(0);
}

///////////

BOOL Dlg_OnInitDialog (HWND hwnd, HWND hwndFocus, LPARAM lParam) {
    chSETDLGICONS(hwnd, IDI_SCHEDLAB);

    // инициализируем классы приоритетов процесса
    HWND hwndCtl = GetDlgItem(hwnd, IDC_PROCESSPRIORITYCLASS);
}

```

Рис. 7-1. Программа-пример *SchedLab*

Рис. 7-1. продолжение

```

int n = ComboBox_AddString(hwndCtl, TEXT("High"));
ComboBox_SetItemData(hwndCtl, n, HIGH_PRIORITY_CLASS);

// запоминаем текущий класс приоритета своего процесса
DWORD dwpc = GetPriorityClass(GetCurrentProcess());

if (SetPriorityClass(GetCurrentProcess(), BELOW_NORMAL_PRIORITY_CLASS)) {

    // эта система поддерживает класс приоритета below normal

    // восстанавливаем исходный класс приоритета
    SetPriorityClass(GetCurrentProcess(), dwpc);

    // добавляем класс above normal
    n = ComboBox_AddString(hwndCtl, TEXT("Above normal"));
    ComboBox_SetItemData(hwndCtl, n, ABOVE_NORMAL_PRIORITY_CLASS);

    dwpc = 0; // данная система поддерживает класс below normal
}

int nNormal = n = ComboBox_AddString(hwndCtl, TEXT("Normal"));
ComboBox_SetItemData(hwndCtl, n, NORMAL_PRIORITY_CLASS);

if (dwpc == 0) {

    // эта система поддерживает класс приоритета below normal

    // добавляем класс below normal
    n = ComboBox_AddString(hwndCtl, TEXT("Below normal"));
    ComboBox_SetItemData(hwndCtl, n, BELOW_NORMAL_PRIORITY_CLASS);
}

n = ComboBox_AddString(hwndCtl, TEXT("Idle"));
ComboBox_SetItemData(hwndCtl, n, IDLE_PRIORITY_CLASS);

ComboBox_SetCurSel(hwndCtl, nNormal);

// инициализируем относительные приоритеты потоков
hwndCtl = GetDlgItem(hwnd, IDC_THREADRELATIVEPRIORITY);

n = ComboBox_AddString(hwndCtl, TEXT("Time critical"));
ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_TIME_CRITICAL);

n = ComboBox_AddString(hwndCtl, TEXT("Highest"));
ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_HIGHEST);

n = ComboBox_AddString(hwndCtl, TEXT("Above normal"));
ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_ABOVE_NORMAL);

```

см. след. стр.