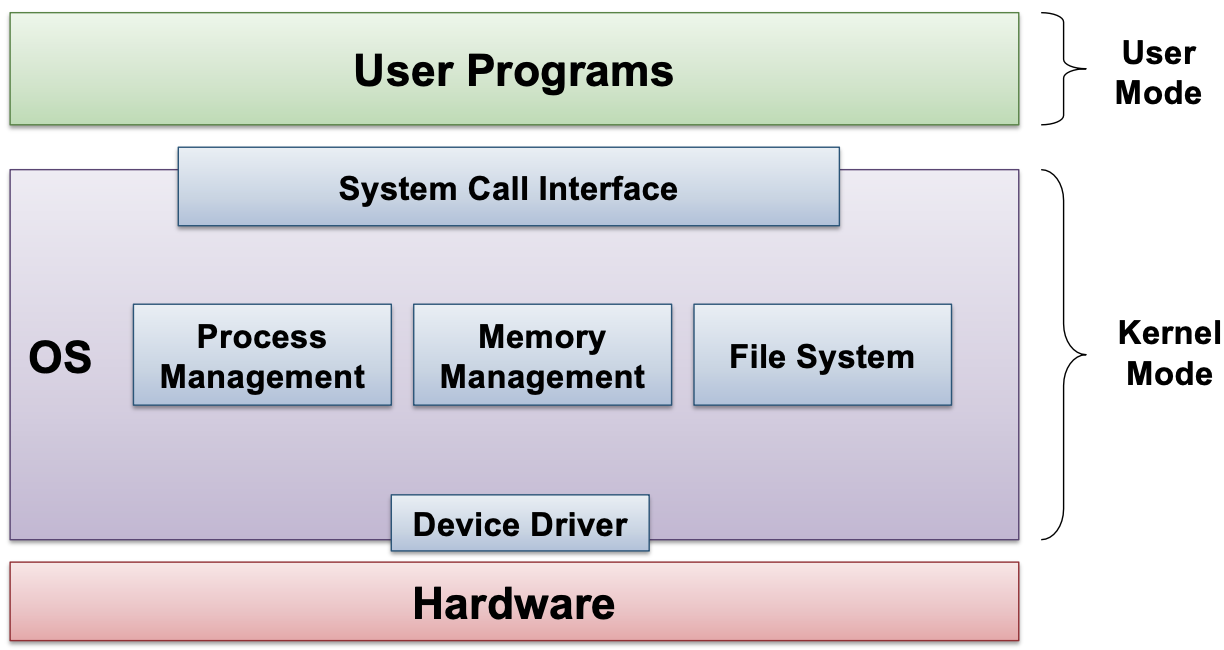
**OS:** A program that acts as an intermediary between a computer user and the computer hardware

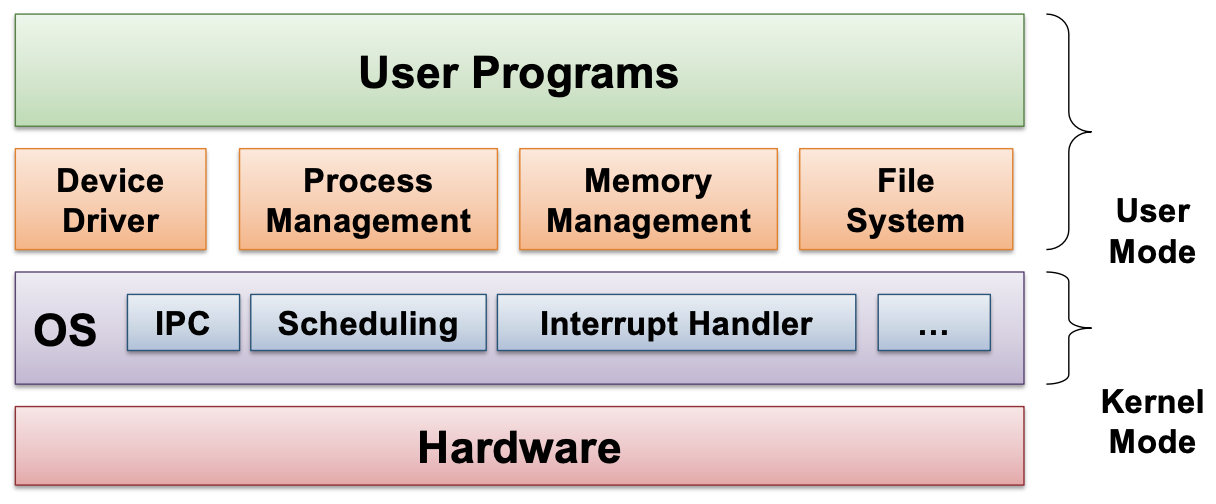
**Monolithic:**

* Kernel is one big special program which includes Sys call interface, process management, memory management, file system, device driver.
* Can load/unload kernel modules on demand, e.g. Linux
* Adv: Well understood & good performance
* Disadv: Highly coupled components & very complicated internal structure

****

**Microkernel:**

* Kernel is very small and clean, only provides basic and essential facilities (e.g. Inter-Process Communication (IPC), address space and thread mgt), everything else is run in user space (e.g. device drivers)
* High level services are built on top of the basic facilities, run as server process outside of the OS, use IPC to communicate.
* Adv: kernel is more robust & extensible, better isolation and protection between kernel and high level services
* Disadv: lower performance



**Type 1 v.s. Type 2 Hypervisors (Virtual Machines)**

| Provides individual virtual machines to guest OSes → implement entire interface to the hardware | Runs in host IS  Guest OS runs inside VM |
| --- | --- |
| More complex to build | simpler |
| faster | slower |
|  |  |

**Process management**

* Abstraction: illusion that process executes on CPU all the time
* Protection: execution context of each process is isolated from each other

**Process Abstraction**

* Includes all information required to describe a running program e.g. Memory Context **(code, data)**, Hardware context **(GPRs, PC, SP, FP)**, OS context **(Process ID & State, resources used)**

**Data and Text:**  Text for instructions i.e. code, Data for global variables

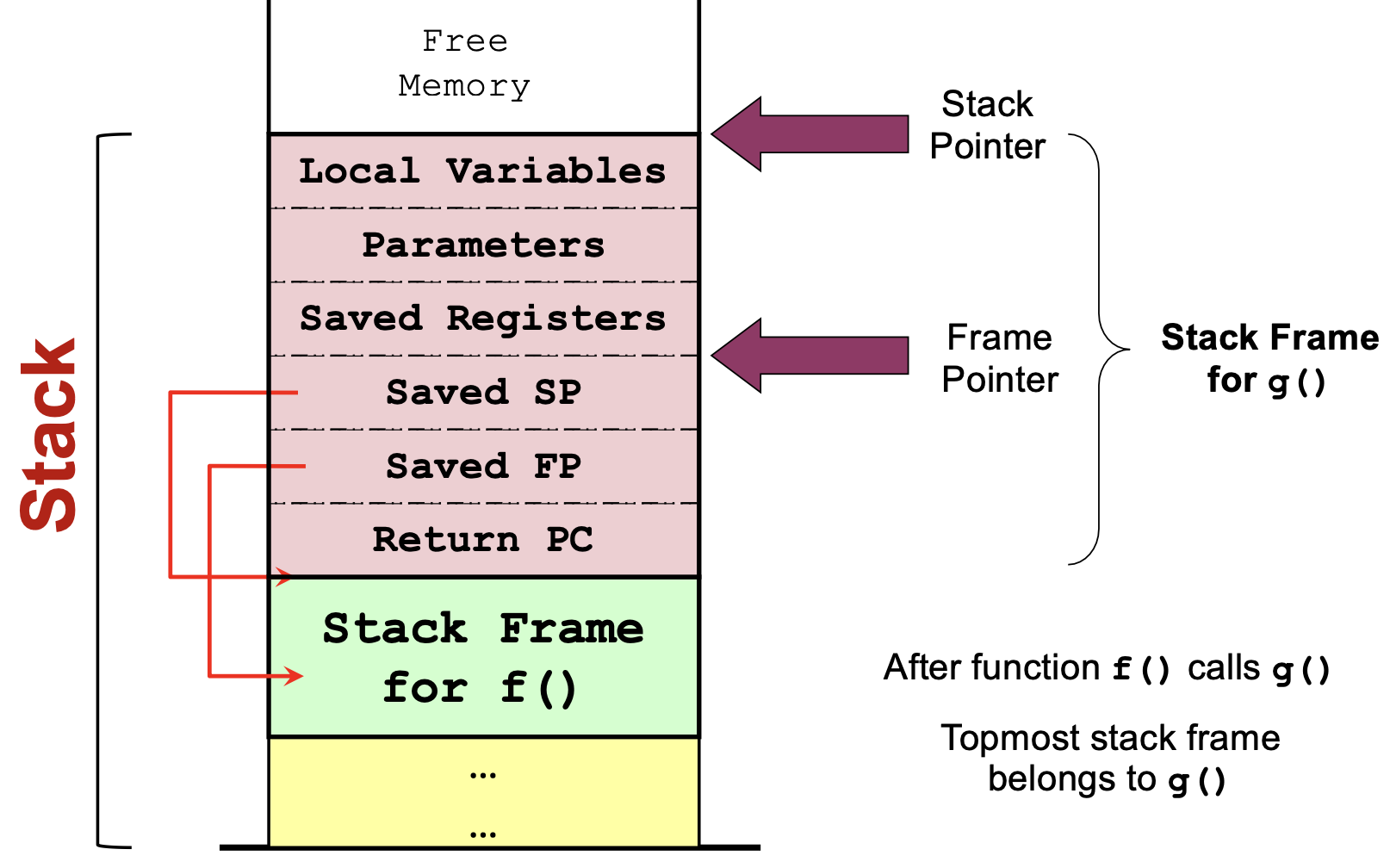
**Stack Memory Region**

Stack frame: RA of caller, Arg of function, Storage for local variables

SP: first unused location

FP: points to a fixed location in a stack frame

Existence of FP&SP depends on the ISA/platform



**Stack Frame Setup/Teardown - BY COMPILER**

On executing function call:

* Caller: pass arg with registers and/or stack, save return PC on stack
* Callee: save registers used by callee. Save old FP, SP; allocate space for local var of callee, adjust SP to new stack top

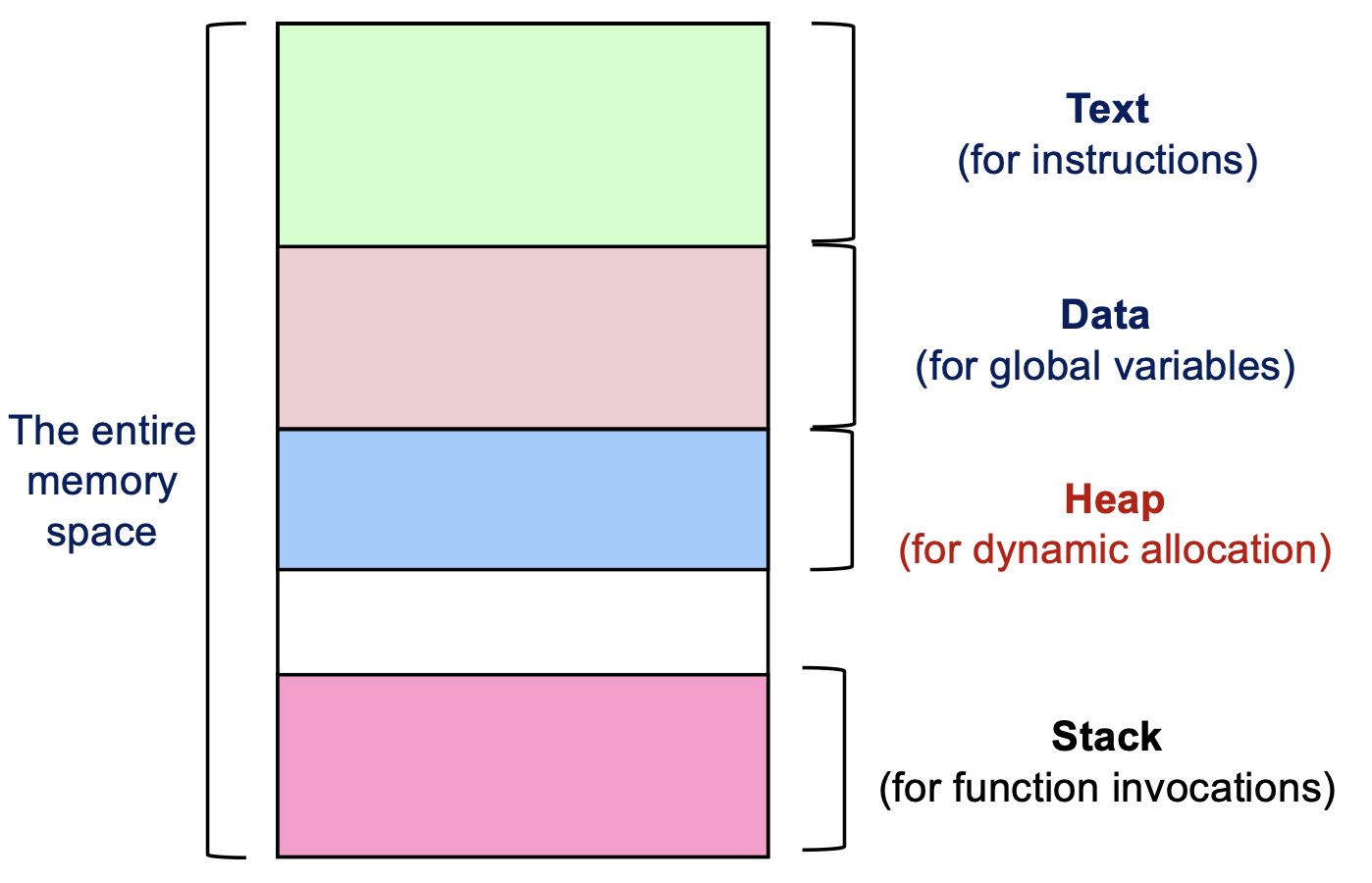
On returning from function call:

* Callee: Restore saved registers, FP, SP
* Caller: continue execution

**Heap Memory**

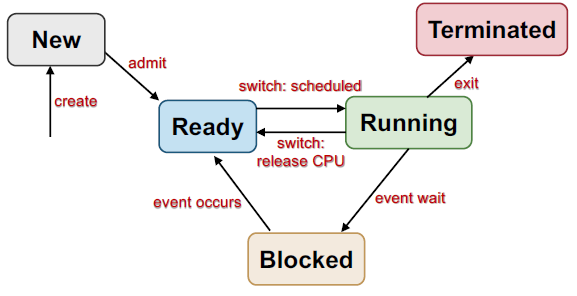
Dynamically allocated memory → memory space during execution time

* Size not known during compilation time so cannot be placed in Data region
* No definite deallocation timing → Cannot be in Stack

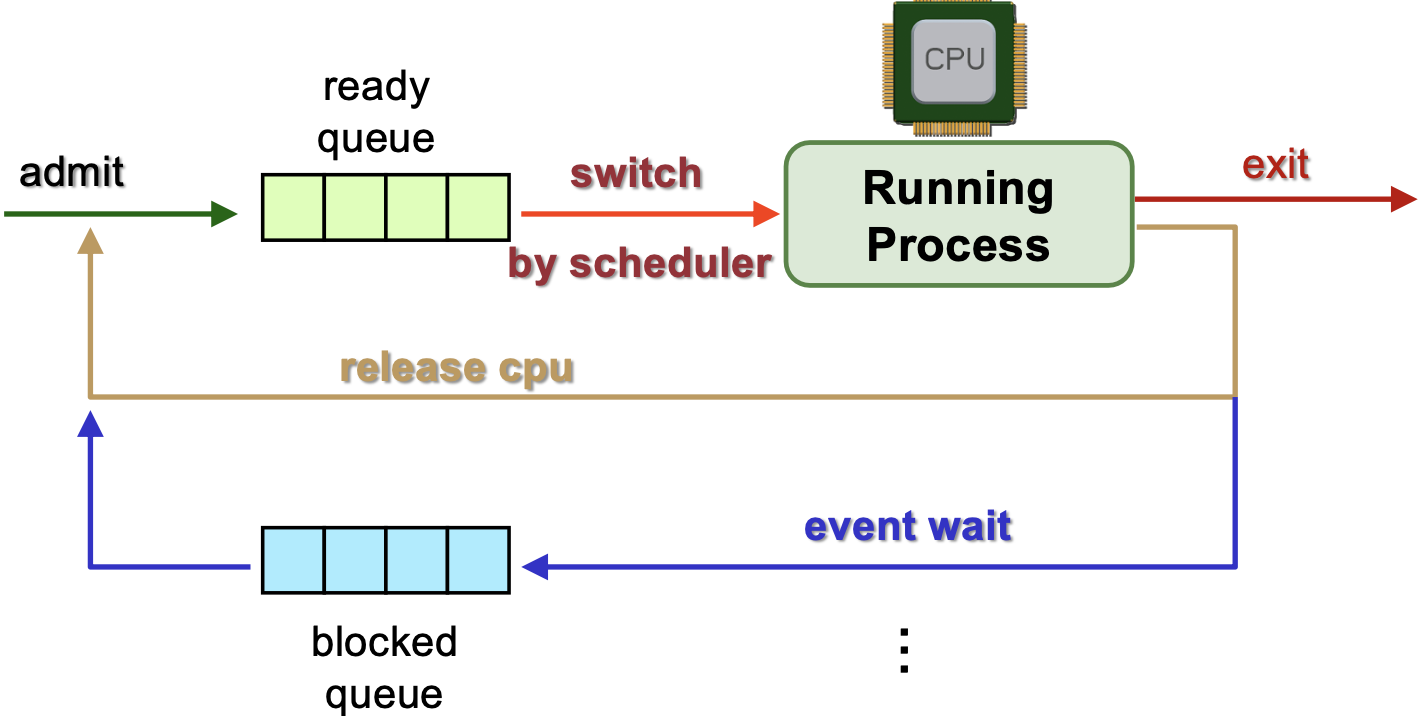


**Process Id & State**

Process Model (incl. states)



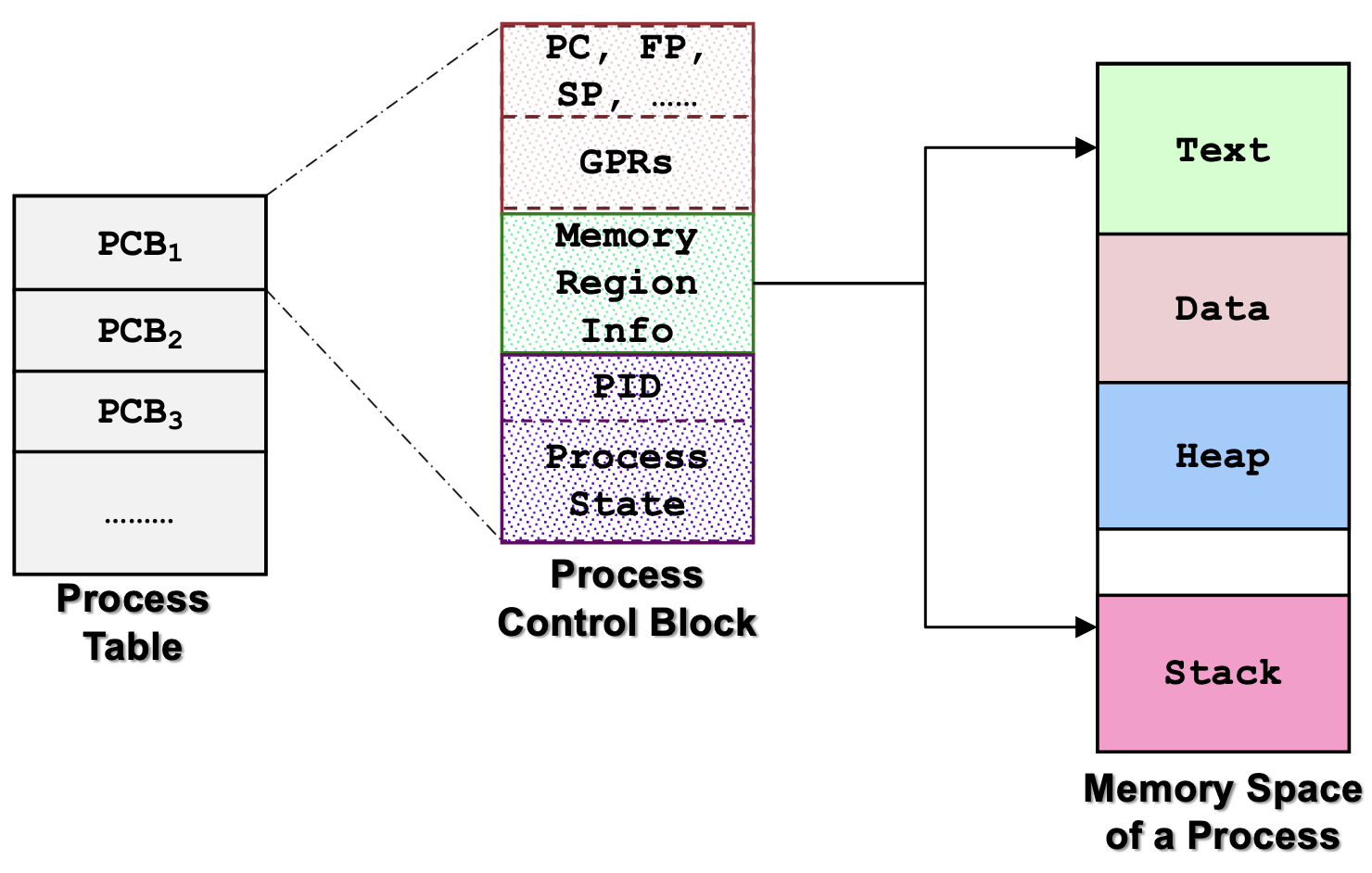
With m CPUs, <= m processes in running state, possibly parallel transitions



“Release CPU” may be due to pre-emption

**Process Table & Process Control Block**

Maintained by kernel

****

* Hardware context in PCB is updated on when process swap out, hence it does not always reflect the actual register values in the processor
* The memory context in the PCB is not the actual memory space used by the process (pointer to real memory space)
* PCBs themselves are stored in the memory
* OS context of a PCB may contain info used for scheduling (e.g. priority, TQ etc)

**System Calls**

* Calling facilities/services in kernel
* Majority of the sys calls have a library version with the name and param (function wrapper), some lib ver are simpler (function adapter).
* Syscalls are invoked by the user programs and libraries, and executed by the OS.
* Transit from user to kernel
* Syscalls are dependent only on the OS while lib calls are programming language dependent
* Adv: hide the low level hardware details from user program, protect system integrity from user programs
* Disadv: requires execution mode change and possibly context change → overhead

General System Call Mech

1. User program invokes the library call
2. Library call places the system call number in designated location e.g. Register
3. Lib call executes special instruction to switch from user to kernel mode (TRAP)
4. Appropriate system call handler is determined
5. Sys call handler executed
6. Sys call handler ended
7. Lib call return to user program

| **Exception** | **Interrupt** |
| --- | --- |
| Happens during the execution of a machine level instruction (e.g. arithmetic, memory access errors) | External events can interrupt the execution of a program  Usually hardware related  (e.g. timer, mouse/keyboard movements) |
| Synchronous | Asynchronous |
| Effects:   * Have to execute an exception handler * Similar to a forced function call | Effects:   * Program execution is suspended * Have to execute an interrupt handler (switch to kernel mode) * Disabling interrupts disable context switch |
|  | Needed to disrupt execution flow, get control of a user, for preemption, for fault handling.  Without interrupts, there will be no quantum-based process switching (because even timer interrupts cannot happen)  CPU handles by checking the interrupt register after finishing the execution of the current inst |

**Process Abstraction in UNIX**

fork() (a syscall)

* Creates new process (child process)
* Child process is a duplicate of the current executable image i.e. same code, same address space
* Data not shared between child and parent
* fork() returns child id if it is parent process; 0 if it is child process
* Child process is irrelevant to the terminal, i.e. if the child is an infinite loop, the terminal can accept new commands after the parent ends.
* fork() n times in a row unconditionally (e.g. fork(); fork(); fork(); fork();), there are 2^n processes in the end, 2n-1 child
* A process that executes fork() must transition from user to kernel mode at some point.

Implementation of fork()

1. Create address space of child process
2. Allocate new PID
3. Create kernel process data structures
4. Create kernel environment of parent process
5. Initialize child process context
6. Memory copy from parent process
   1. Copy on Write - only duplicate memory location when either parent/child writes, otherwise they share the same memory
7. Acquire shared resources
8. Initialize hardware context for child
9. Child ready to run

exec()

* Replace current executing process image with a new one, use after fork()
* Only code replacement, PID and other information still intact
* causes a switch from user to kernel mode

Master (root) Process

* init process - created in kernel at boot time
* PID = 1
* Watches for other processes and respawns where needed

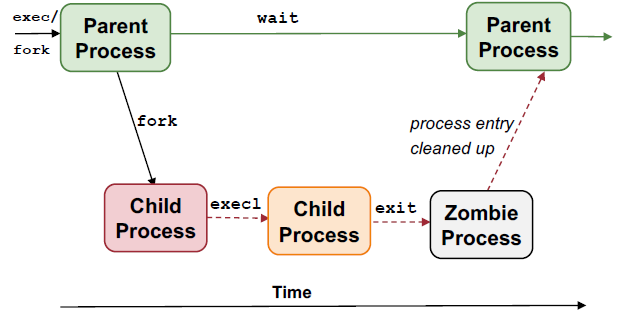
Process Termination

* Most system resources used by process are released on exit → syscall to notify OS to clean up
* Not releasable process resources incl. PID & Status, Process accounting info, PCB
* Child exit() sends SIGCHILD to parent, SIGTERM sent by KILL

Parent-Child Synchronization

wait(): returns PID of the terminated child process

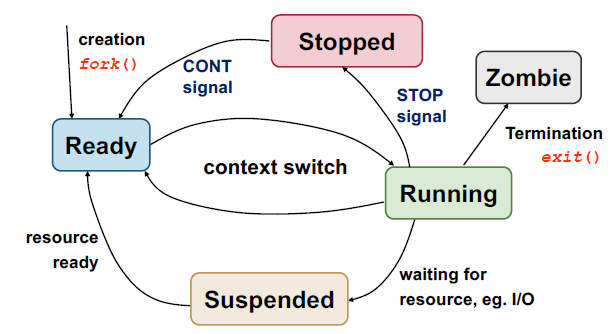
* Blocking → Parent process blocks itself until at least 1 child terminates
* Cleans up child system resources e.g. zombie process & Removes child from Process Table
* Parent exit before child → C orphan



Zombie Process

1. Parent process terminates before child process
   1. init becomes “pseudo” parent of child
   2. Child termination sends signal to init, which utilizes wait() to cleanup
2. Child process terminates before parent but parent did not wait() → C zombie → fill up Process Table → may need roboot

* Cannot delete all process info
* Cannot kill zombie process
* Zombie process takes up a slot in the OS PCB table
* Zombie process is created so that wait() can be implemented properly (zombie return info about the terminated child process)
* A user command running in the background under a shell can become a zombie process
  + A foreground process is waited by the shell interpreter directly while a background process is not waited → becomes zombie if terminated



* Stopped: blocked by other processes

CPU-bound processes (aka Compute-bound)

* CPU-intensive

IO-bound processes

* IO intensive → needs to be more responsive

Waiting time = TAT - Time required

Response time = first time the task gets CPU - arrival

**Types of processing environment**

1. Batch processing
   * Usually long-running without user intervention → no need to be responsive
   * Process are still allowed to do I/O operations
   * Non-preemptive scheduling is predominant: FCFS, SJF, SRT
   * Criteria
     1. Throughput: num of tasks finished per unit time
     2. Turnaround time (TAT): finish time - arrival time
     3. CPU utilization: % of time when CPU is working on a task
2. Interactive
   * Should be responsive, consistent in response time (e.g. MS word)
   * Criteria
     1. Responsive
     2. Predictability: less variation in response time → better predictability
3. Real-time processing
   * Have a strict ddl to meet (e.g. aircraft controller)
   * Tasks are usually periodic

**Mandatory criteria for Scheduling Algo**

* Fairness:
  + fair share of CPU time (on a per-process or per-user basis)
  + no starvation (guarantee that the process will be ran)
* Balanced utilization of the system resources
  + All components of the computing system should be utilized in a balanced manner, without bottlenecks

Non-preemptive (aka cooperative)

* A process stayed scheduled (RUNNING) until it blocks (BLOCKED) or gives up the CPU voluntarily (READY)
* Can cause starvation

Preemptive

* CPU can be taken (preempted) from the running process at ANY time
* A process is given TIME QUOTA to run
* At the end of TQ, the process is suspended (READY)
* A process can block or give up CPU

**Scheduling a Process**

1. Scheduler is triggered (OS takes over)
2. If context switch is needed (e.g. TQ expires), context (PC, Reg, SP) of current running process is saved and placed on blocked queue/ready queue
3. Pick a suitable process P to run based on scheduling algo
4. Setup the context for P
5. Let P run

**Periodic Schedule Invocation**

* OS ensures time interrupt cannot be intercepted by any other program (or interrupt)
* Time interrupt handler invokes the scheduler
* Interval of TIme Interrupt (ITI): time period at which OS scheduler is triggered (~1-10ms)
  + Shorter ITI → more timer interrupt → less time spent on actual user process → lengthen TAT
* Time quantum (TQ): execution duration given to a process, could be const or var among the processes, must be multiples of ITI (~5-100ms)
* Shorter TQ → more context switches → more overhead due to context switching and longer TAT but process becomes more responsive

[Non-Pre] First-Come-First-Served (FCFS)

* Pick the 1st task in queue to run until
  + The task is done or blocked
  + Blocked tasks is removed from the FIFO queue, placed at the back of the queue when it is ready to run
* No starvation: num of tasks before X is decreasing, X will be able to run eventually
* Shortcoming: convoy effect - IO bound processes delayed by CPU bound processes in front

[Non-Pre] Shortest Job First (SFJ)

* Select task that need the shortest amount of CPU time
* Need to know the total CPU time in advance → predict (exponential average approach)
  + Predictedn+1 = α Actualn + (1-α) Predictedn
* Guarantees smallest average waiting time
* Possible starvation: bias towards short jobs, long jobs might never get a chance to run
* Max throughput: max #jobs completed per unit time

[Pre] Shortest Remaining Time (SRT)

* New jobsz with shorter remaining time can preempt (replace) currently running job
* If all tasks arrive at the beginning, SRT will give the same schedule as SJF.
* Possible starvation

[Pre] Round Robin (RR) - most suitable for time-shared OS

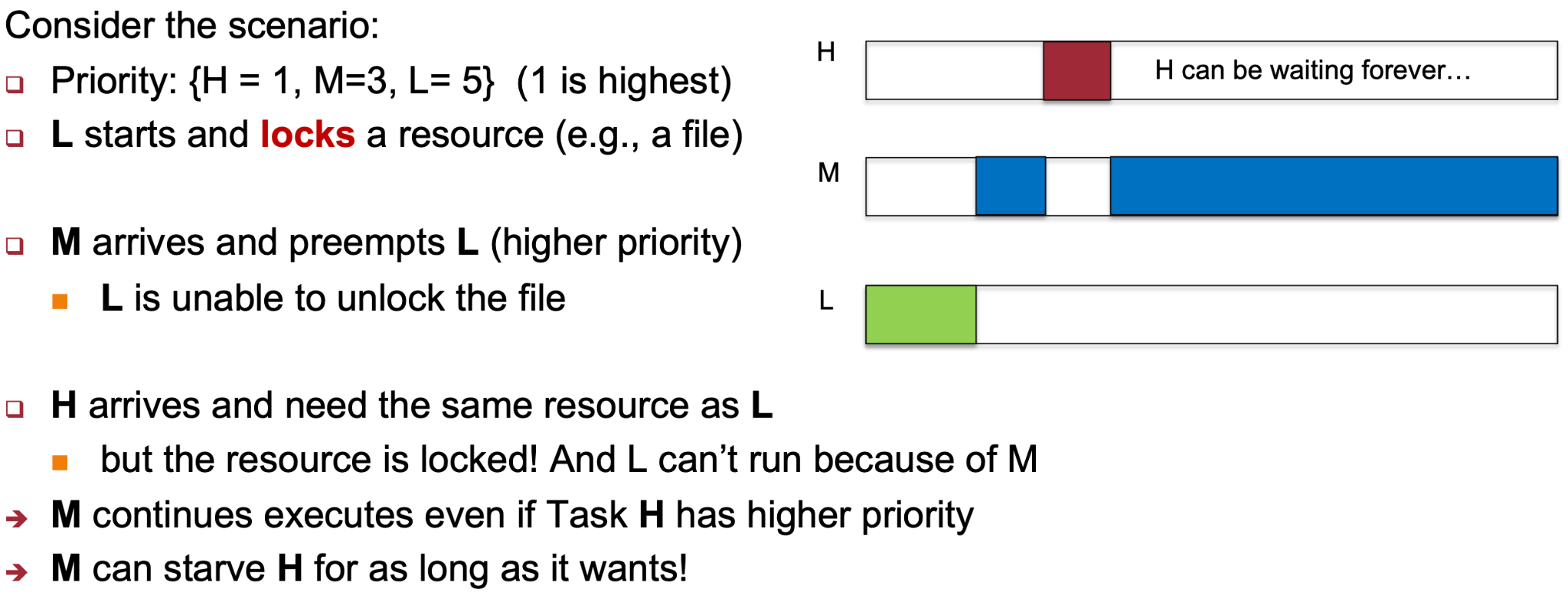
* Tasks are stored in FIFO queue
* Pick the first task from queue front to run until
  + The task gives up the CPU voluntarily or
  + The task blocks or
  + A fixed time slice (TQ) has elapsed (key diff from FCFS)
* The task is then placed at the end of the queue to wait for another turn
* Adv: guarantees responsive time → worst waiting time = (n - 1)q
* Timer interrupt needed for scheduler to check on quantum expiry
* Same as FCFS if TQ is infinitely large (> job lengths)
* Perform poorly than FCFS if the job lengths are the same and >> TQ (more context switches and overhead)
* Big TQ: better CPU utilization but longer TAT
* Smaller TQ: bigger overhead (worse CPU utilization) but shorter waiting time → more responsive

[Non-Pre/Pre] Priority scheduling

* Assign priority value to all tasks and select the one with highest priority value
* [Non-Pre] late coming high-priority process has to wait for next round for scheduling
* [Pre] higher priority process can preempt running process with lower priority
* Hard to guarantee the exact amt of CPU time for a process
* Possible starvation for lower priority processes, even worse in preemptive ver
* Possible solutions:
  + Decrease the priority of currently running processes after every TQ
  + Give each process a min TQ - ensures that low-priority processes gets to run at least for a while

Priority inversion: Lower priority task effectively preempts higher priority task

* A system with all jobs having exactly two priorities (either H/L) does not suffer from the priority inversion.



Solution: Priority inheritance

* Low-priority job inherits the priority of the higher priority process when the high priority process requests a lock held by low-priority process → Priority restored upon unlock

[Pre] MLFQ

Basic Rules

* Prioirity(A) > Priority(B) → A runs
* Priority(A) == Priority(B) → A&B runs in RR

Priority Setting/Changing rules

* New job → highest priority
* If a job fully utilized its time slice → priority reduced
  + CPU-intensive tasks are penalized in priority
* If a job give up/blocks before it finishes the TQ → priority retained
  + Potential to abuse the algo by always giving up before TQ → hog CPU & get unfair share of CPU → mitigate by using cumulative time
* Lower priority usually have longer TQ
* Favours IO intensive processes as their CPU time is likely to be < TQ hence they can retain their priority and be more responsive
* Favours shorter jobs
* Same as RR is there is only 1 priority level

Lottery Scheduling

Scheduling done in round, in every round

* Give out “lottery tickets” to processes
* When a scheduling decision is needed, a lottery ticket is chosen randomly and the winner is granted the resource: all processes will run, lottery determines the order and resource %
* In every round, a process holding X% of tickets can win X% of the lottery and use the resource X% of the time
* Responsive
  + Every participating process gets to run in every round (no starvation)
  + A newly created process can participate in the next lottery round
  + The response time of any process is decided by the duration of the lottery round, which is independent of the order in which processes run within a round, and therefore, independent of the randomness.
* Provide good level of control
  + A process can be given Y lottery tix which can then be distributed to its child
  + Impt process can be given more tix
  + Each resource can have its own set of tix w diff portion of usage per resource per task
  + Simple implementation

**Inter-Process Communication (IPC)**

For cooperating processes to share information

Common IPC mechanisms:

Shared-Memory

1. P1 creates a shared memory region M
2. P2 attaches M to its own memory space
3. P1 and P2 can now communicate using M through read/write
4. Detach M from memory space after use
5. Destroy M (P1)

Advantages: Efficient only need to setup M, ease of use (just read and write)

Disadvantages: Limited to single machine, requires synchronization (race conditions)

* Shared memory region created by program P can stay around after P exited (when theres no explicit delete command - shmctl)
* Shared memory region is identified by a number (id). A mem addr is generated only when it is attached to a process
* Shared memory region can be accessed by any process (including process from other users) (when permission bits are set to allow other processes to access)

Race Condition (aka data races)

In a concurrent system, when a set of processes access some shared resources to carry out a computation and the results of the computation depend on the exact way the processes interleave, there is a race condition.

E.g. 2 processes write to a variable concurrently; one write one read same variable

Necessary conditions for race condition:

* Multiple concurrently executing processes/threads.
* Modifiable resource shared by multiple threads/processes.

Message Passing

Explicit communication through exchange of messages

1. P1 prepares a message M and send it to P2
2. P2 receives the message M
3. Message sending & receiving usually provided as sys calls

Message have to be stored in kernel memory space

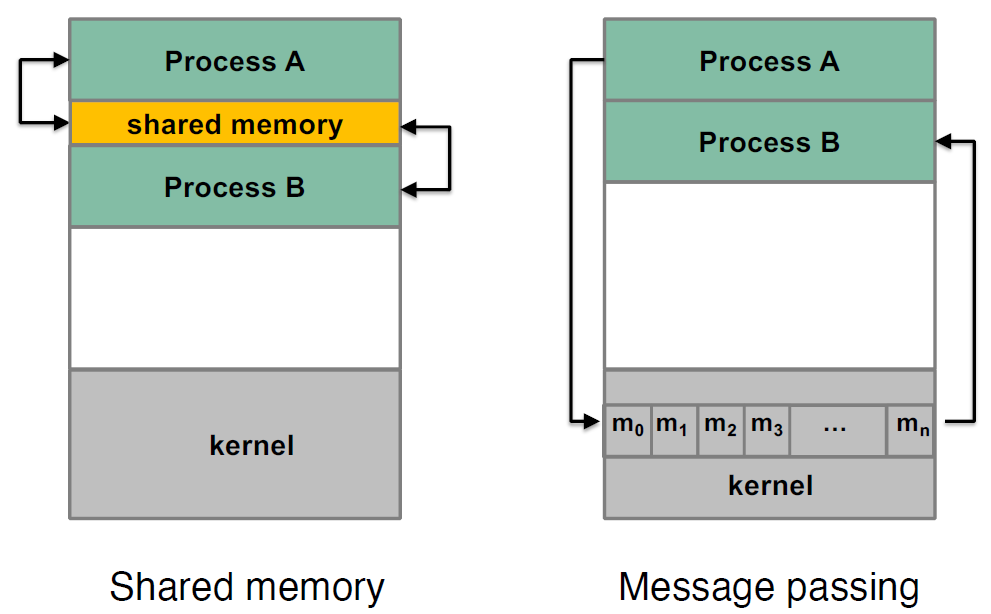
Naming Scheme

Direct Communication:

* Sender/Receiver explicitly name the other party
* 1 link per pair of communicating processes

Indirect Communication:

* Messages are sent to/received from message storage aka mailbox or port (kernel memory)
* 1 mailbox can be shared among a number of processes
* No race conditions happen w msg passing, OS takes care of its own msg integrity
* If mailboxes are read simultaneously, OS allows an arbitrary process to get the msg



| **Shared Memory** | **Message Passing** |
| --- | --- |
| Simple, just read and write (accessed similarly to a normal mem location through load and store) | Complex. Need to have mailbox as proxy. |
| Only available on single machine | Flexible and Applicable → more general and portable, applicable beyond a single machine |
| synchronization (by the programmer) is required | Easier synchronization |
| OS only involved in setup | Requires OS intervention upn every send/receive |

Synchronization Behaviors

Blocking Primitives (Synchronous):

* Sender is blocked until the message is received
* Receiver is blocked until a message has arrived

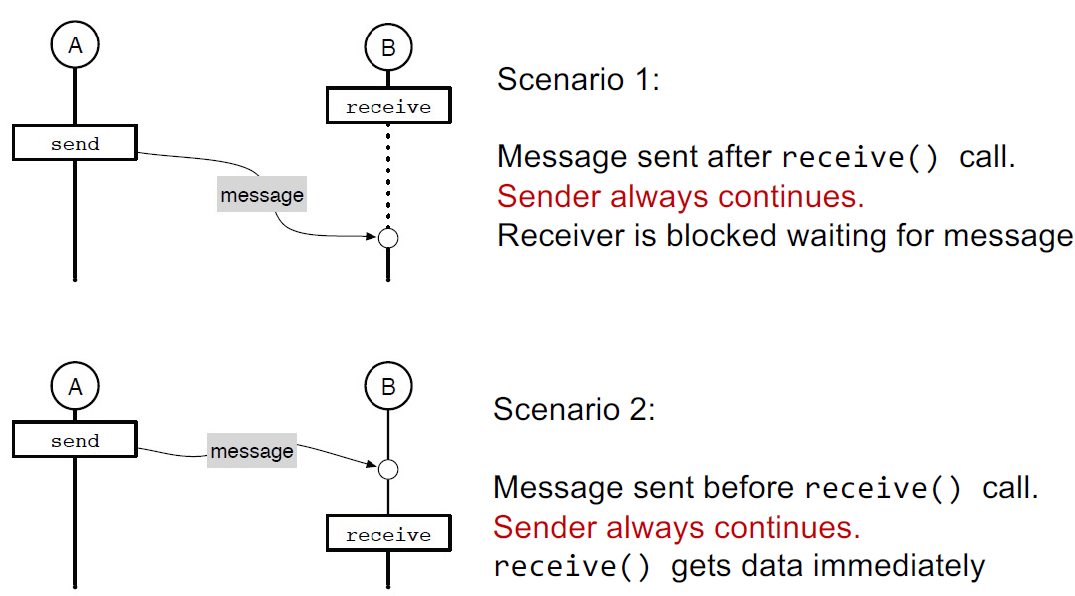
Non-blocking Primitives (Asynchronous):

* Sender resumes operation immediately
* Receiver: if message hasn’t arrived yet, proceeds empty-handed but no block

**CS2106 ASSUMES BLOCKING RECEIVE!**

Asynchronous Message Passing

* Sender is never blocked even if receiver has not yet executed the matching receive(), unless buffer full
* System buffers the message and complete receive() later

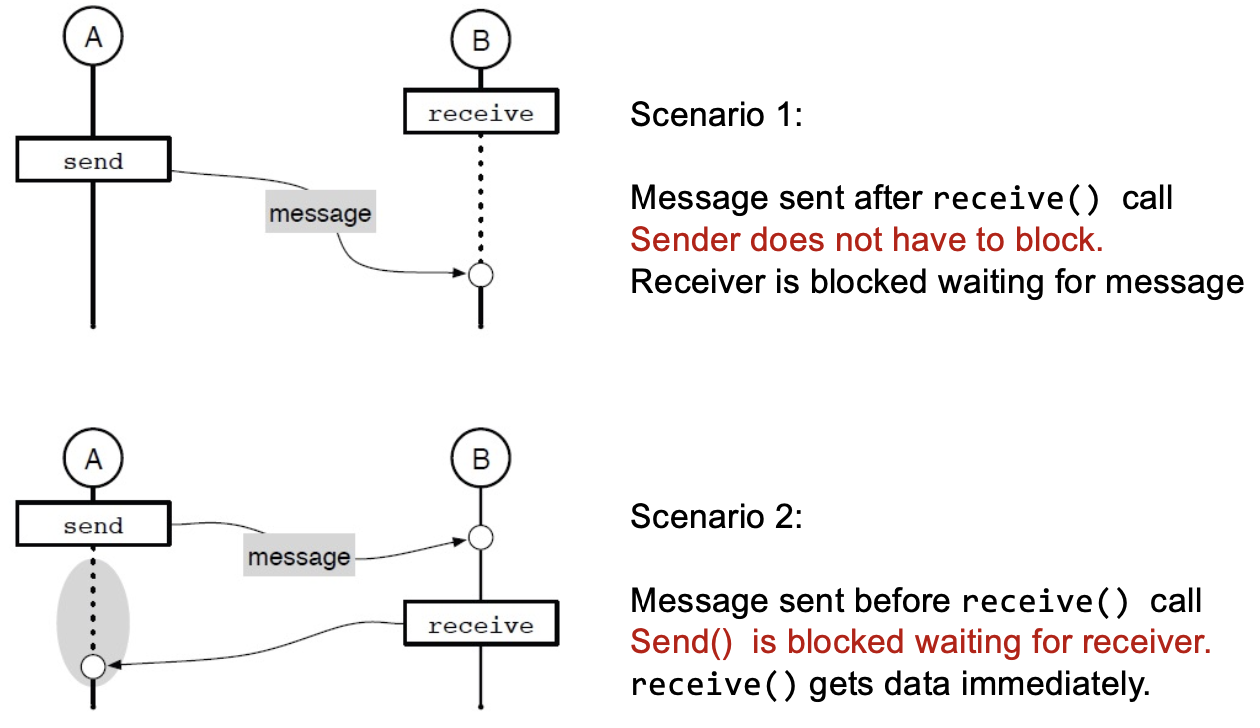


Message Buffers:

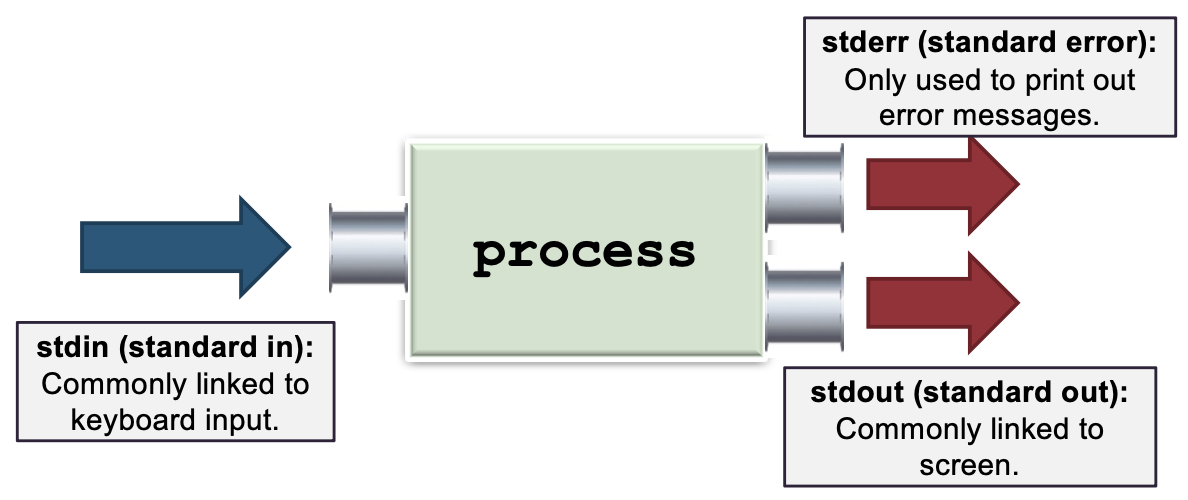
* Under OS control
* Large buffer makes sender and receiver less sensitive to variations in execution
* User may need to declare in advance the capacity of the mailbox when mailbox is created
  + Sender blocks when buffer full OR
  + Sender returns immediately with an error

Sync Message Passing (blocking send, aka rendezvous)

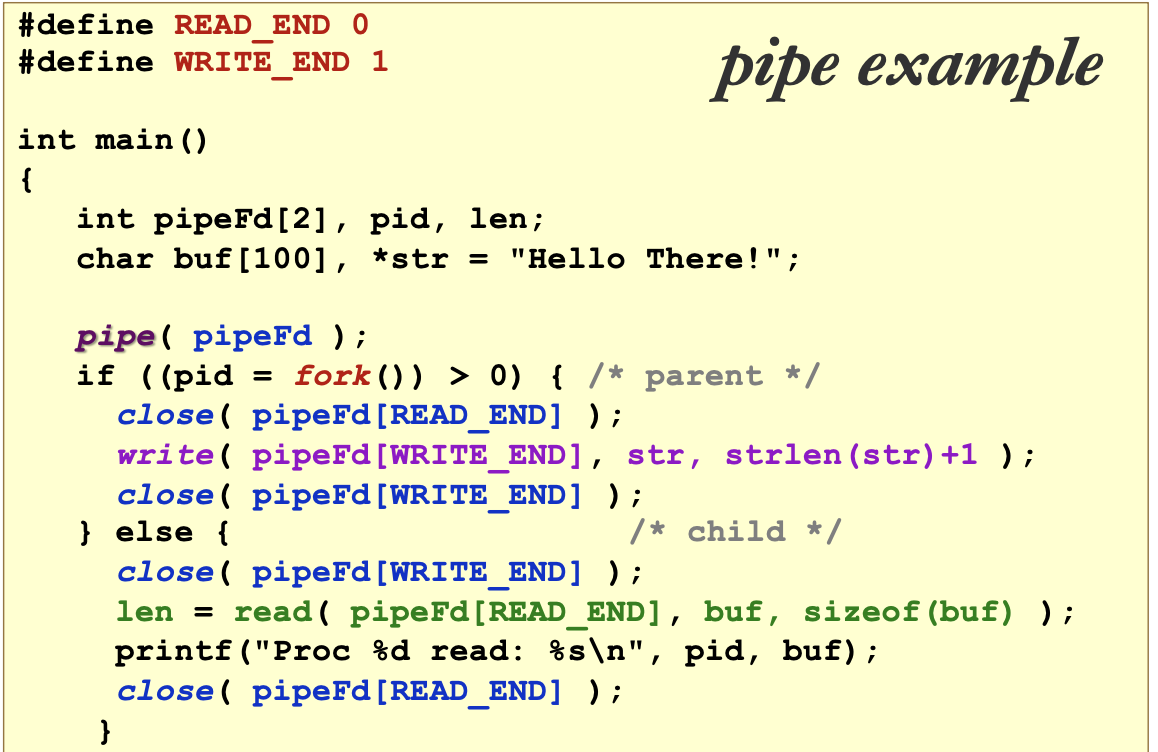
* Sender forced to wait until receiver receive()
* No buffering required: msg kept by sender till receiver calls and then directly copied into the receiver address space



[Unix IPC Mech] Pipes (communication channels)



* 3 default comm channels
* “A | B”: output of A goes to B (instead of screen) as input
* Async msg passing
* Unidirectional (half-duplex): 1 reading (fd[0]), 1 writing (fd[1])
* A pipe can be shared between 2 processes
* FIFO: must access data in order
* Pipe functions as circular bounded byte buffer w implicit synchronisation
  + Writes wait when buffer is full
  + Readers wait when buffer is empty
* Variants:
  + Multiple writers/readers
  + Bidirectional (full-duplex): any end for read/write



[Unix IPC Mech] Signals (event notifications)

* An asynchronous notification regarding an event sent to a process/thread
  + A program can receive a signal from OS anytime during its execution
* Recipient of signal must handle the signal by:
  + A default set of handlers OR
  + User supplied handler (only applicable for some signals)
* Common signals in Unix: Kill, Stop, Memory Error, Arithmetic error, etc.
* A process can install user-define handler for multiple different signals
* Only some signals can have user-define handlers
* A parent process cannot force the child processes to execute any part of their code by sending signal to them. Only the signal handler can be triggered.
* The "kill" signal (sent by the “kill” command) is different from the “interrupt” signal (sent by pressing ctrl + c)
  + KILL command is more versatile, it can send any possible signal to any process, e.g. SIGKILL, SIGTERM, SIGQUIT, SIGINT etc
  + Interrupt only sends SIGINT
* A signal handler is a special routine that is executed when a signal is received by a process.
* A signal handler cannot wait or signal a semaphore. (sem\_wait is not async\_safe)

**Process Alternative - Threads**

Motivation:

1. Process is expensive
   * Process creation: Duplicate memory space and most of the process context
   * Context switch: requires saving/restoration of process information
2. IPC is challenging and inefficient
   * Independent memory space

Threads in the same process share:

* Memory context: Text, Data, Heap, incl page tables (virtual mem state) and file descriptor table
* OS Context: PID, other resources e.g. files

Threads do not share (Hardware context:

* PC: each threads maintains its own execution
* GPR: threads share the same GPR hardware but they have their own GPR values that are private to themselves. OS make sure GPR values are loaded correctly
* Stack: manage indiv control flow

Benefits:

1. (economy) Multiple threads in the same process requires much less resource to manage compared to multiple processes
2. (resource sharing) Threads share most of the resources of a process - no need for IPC mech
3. (responsive) Appear more responsive (new process can get its first share of CPU time quickly)
4. (scalable) Can take advantage of multiple cores

Problems:

1. Synchronization around shared memory gets even worse - all memory except stack is shared
2. System call concurrency
   * Parallel execution of multiple threads → concurrent sys call invocations possible
3. Process Behavior - Impact process operations
   * E.g. which thread receives a signal sent to the process

| **User Threads** | **Kernel Threads** |
| --- | --- |
| Implemented as a user library: a runtime system (in the process) handles thread-related operations (library calls) | Implemented in the OS: thread operations are handled as syscalls |
| Kernel views all threads as only one thread → scheduling performed at process level → can only execute on one core  one thread blocked, all blocked | Kernel schedule threads, not processes → can execute on multiple cores  if one thread is blocked, other threads of the same process can still run |
| Can have multithreaded program on ANY OS | Syscalls are slower and more resource intensive |
| More configurable & flexible | Less flexible  Used by all multithreaded programs |
| Both need hardware support | |

Hybrid Thread Model

* User thread bind to kernel thread
* More flexible → limit concurrency

Threads on Modern Processor

* Simultaneous multi-threading (SMT)
* Supply multiple sets of registers to allow threads to run natively and in parallel on the same core

**Process Management - Synchronization**

Race Condition Solution:

* Designate code segment with race condition as critical section
* At any point in time, at most one process can be in the critical section

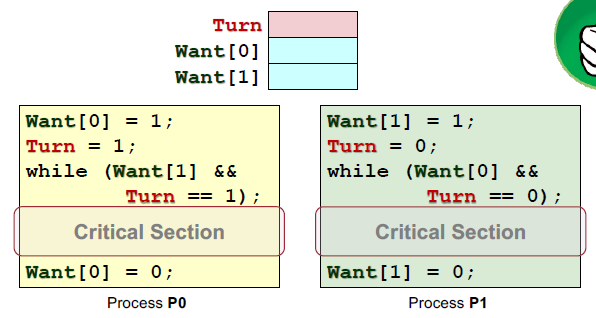
Properties of Correct CS Implementation

1. [Mutual Exclusion] If process P1 is executing in CS, all other processes are prevented from entering CS
2. [Progress] If no process is in CS, one of the waiting processes should be granted access
3. [Bounded Wait] After Process P1 request to enter CS, there exists an upper bound on the number of times other processes can enter the CS before P1 (guaranteed no starvation)
4. [Independence] Process not executing in critical section should never block other processes

Incorrect Synchronization

* Incorrect output/behavior
* Deadlock: All processes blocked & no progress
* Livelock:
  + Typically processes are not blocked
  + Processes keep changing state to avoid deadlock but make no other progress
  + Maybe able to get out of livelock by chance
* Starvation: Some processes are blocked forever

Peterson’s Algorithm

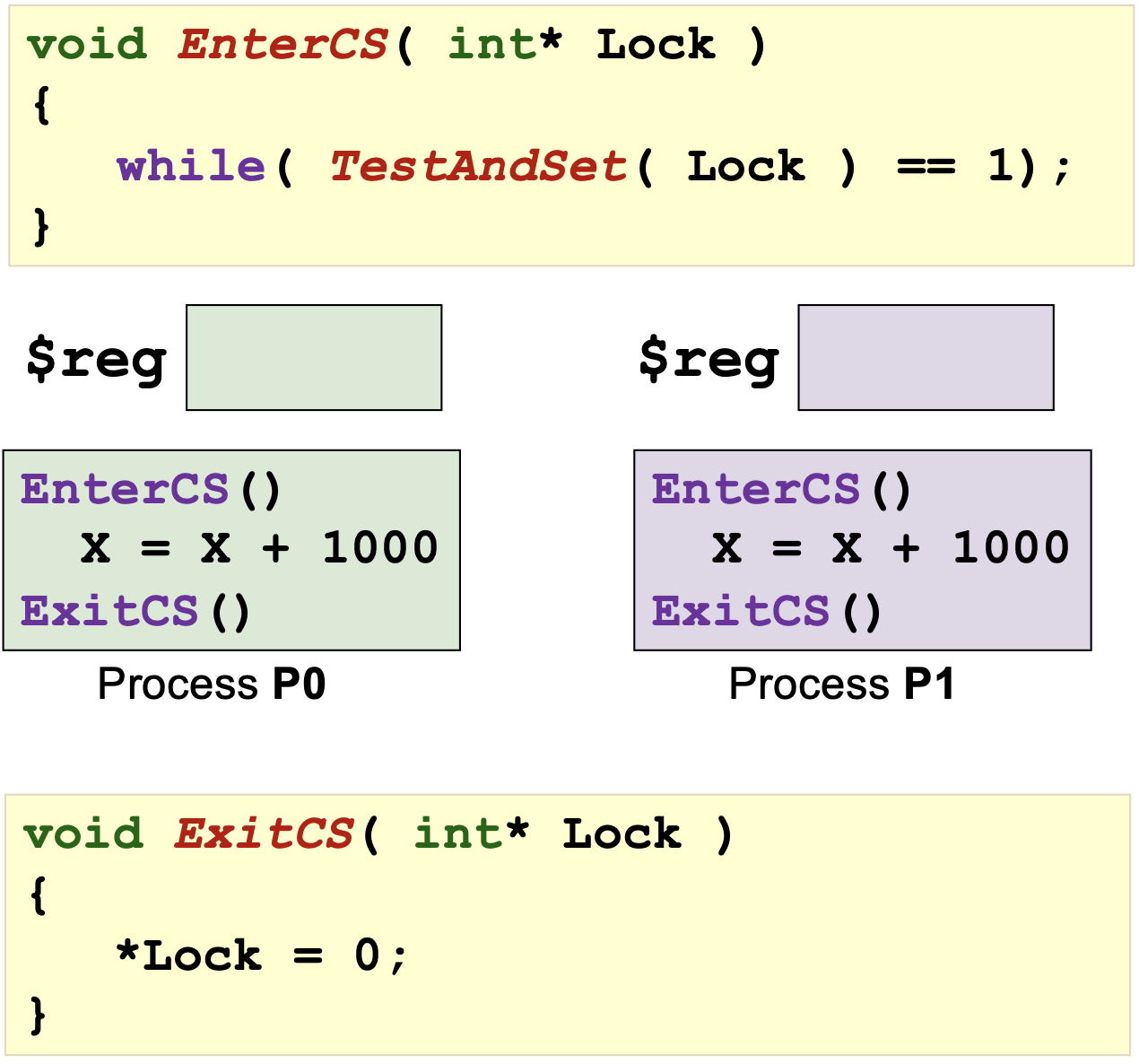


Disadvantages:

1. Busy Waiting
   * Waiting process repeatedly test the while-loop condition instead of going into blocked state
2. Low level
   * Higher-level programming construct is desirable → simpler and less error prone
3. Not general
   * General synchronization mechanism is desirable

Test and Set (atomic instruction)

* Exchange data between mem addr & a reg
* Load the current content at MemoryLocation into Register
* Stores a 1 into MemoryLocation (act as a lock)

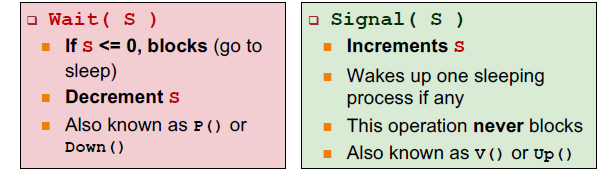


* Cons: busy waiting, does not guarantee bounded-wait unless the sacheduling is fair
* Variants: compare and exchange, atomic swap, load link/store conditional

High Level Synchronization Mechanism

Semaphore:

* A generalized synchronization mechanism
* Only functional behavior is specified → can have different implementation
* Provides means to:
  + Block a number of processes aka sleeping processes
  + unblock/wake up one or more sleeping processes
* Contains an integer value S → can be initialized to any non-negative values initially
  + Protected integer
  + List to keep track of waiting processes
* 2 atomic operations Wait() and Signal()
* Incorrect use of sem can still cause deadlock, violate mutual exclusion.
* A task can be blocked on two sem simultaneously



Properties:

Given Sinitial >= 0

Scurrent = SInitial + #signal(S) - #wait(S)

#signal(S)/wait(S): num of signal(S)/wait(S) completed

General and Binary Semaphores

General aka counting semaphores:

* S >= 0 (S = 0, 1, 2, 3, …)
* Convenience

Binary Semaphore:

* S = 0 or 1
* Sufficient

Mutex: Correct CS

NCS = Number of process in critical section

= Process that completed wait() but not signal()

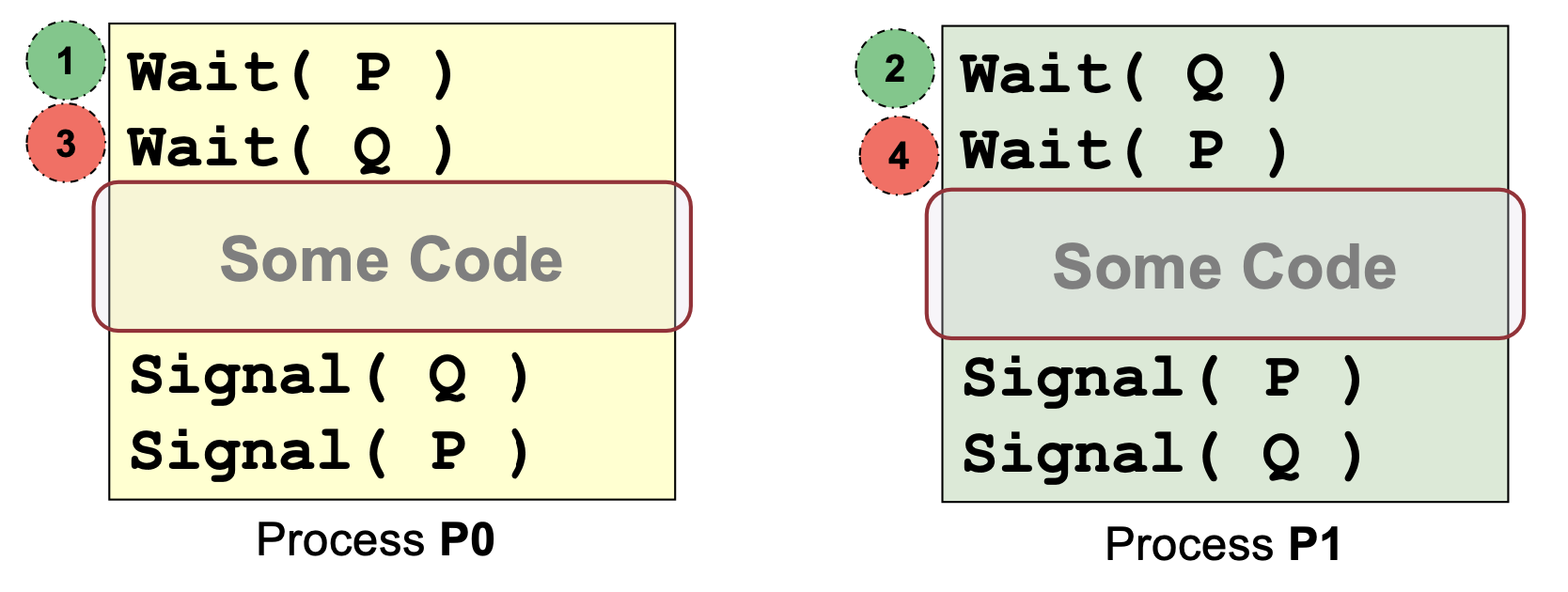
= #Wait(S) - #Signal(S)

Sinitial = 1

Scurrent = 1 + #Signal(S) - #Wait(S)

Scurrent + NCS = 1

Incorrect use of semaphore: Deadlock



Context switch after 1 → P&Q wait for each other

Use Semaphore to synchronize

* Execute B in P1 only after A executed in P0

S(0)

P0 P1

A wait(S) // blocks B from executing

signal(S) B

**Exam Questions**

[1112s1] Guidelines to implements tasks as processes or as threads

Threads > process:

* Large num of tasks
* Tasks share much common code
* Much communication is needed among the tasks
  + all threads w/i a process share the same address space, need not use shared memory.
  + Inter-thread communication facilitated by thread libraries

Process > threads:

* Tasks involves executing potentially buggy code
* More protection, isolation, security
* When large memory space is needed
* Better synchronization
* Non-thread safe syscalls
* *User* threads could result in blocking for the other *user* thread

[1112s1] A&B use the same RR scheduler but A has a fast I/O subsystem

* A has higher CPU utilization, higher throughput, lower TAT
* A’s processes spend long time waiting in queue because they join the queue faster

[1516s1]

While (fork() != 0) {

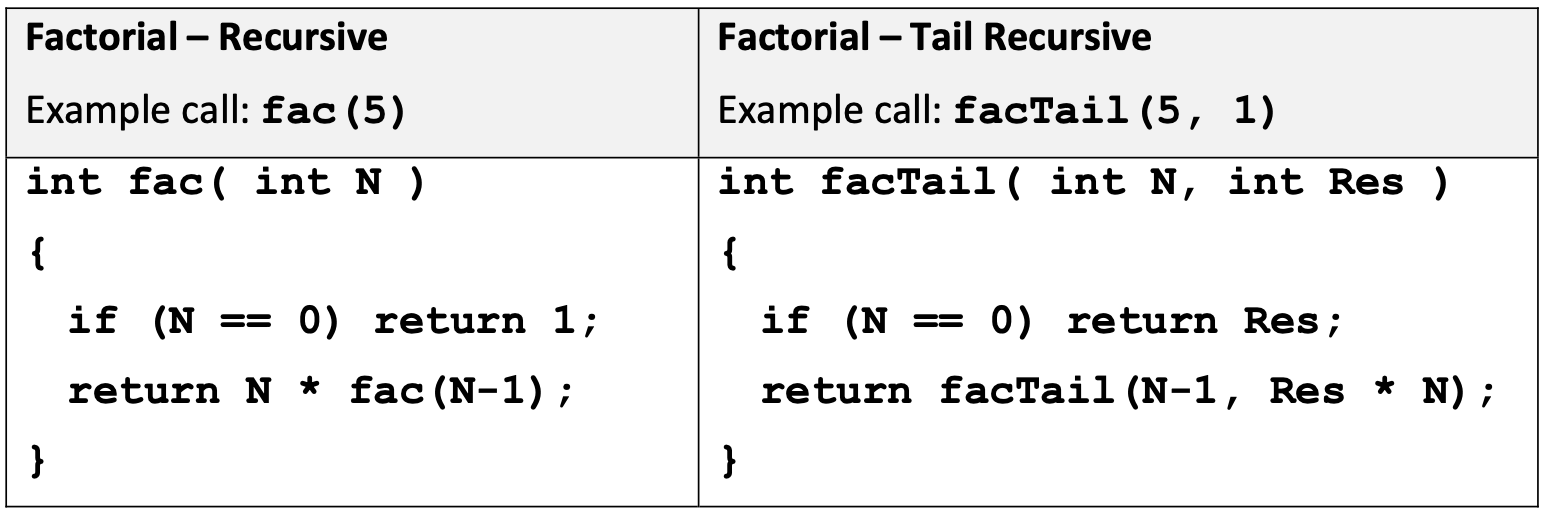
execl(“Hello.exe”, “Hello.exe”, NULL);

}

* Only when “Hello.exe” is not a valid executable, then the code is a “fork bomb”
  + execl() will replace the calling process with the executable image if the exectable can be found
  + Invalid executable will cause the execl() to fail and returns normally → loop again

[1617s1] Minimal set of info to keep for a zombie process to support wait() correctly: PID, return value

[1617s1] How can we exploit the behaviour of tail-recursive function call to reduce the usage of stack memory?



* Stack frame for a function only needs to be maintained if it is going to continue execution.
* In tail recursion, once a function make a recursive call, it ceased to be useful.
* Exploit by allowing the recursive call to overwrite the current stack frame instead of allocating a new frame ⇒ max frame = 1

[1718s1] Suppose process A is current executing and is given 2 time units time quantum to run:

| 0 | 1 | 2 | 3 |
| --- | --- | --- | --- |
| **A** | **A** | … | … |

New process B only arrives in the shaded region. In unix, process can be created only via fork() / clone() mechanism → A has to invoke one of theseonly when A is executing.

Responsive (short waiting time) scheduling algo: Lottery, MLFQ, Priority

Irresponsive scheduling algo: FCFS, RR (new tasks always queued at the end)

[1819s1] Semaphore P(1), Q(1)

| Task A | Task B |
| --- | --- |
| wait(P);  …  signal(Q); | wait(Q);  …  signal(P); |

ensures that A&B enter CS in strict order: A → B → A → B

If the above code is used for a scenario where A&B can enter CS in any order, which CS criteria is/are violated?

* Independence: Task B can be blocked if it reaches the critical section first even though A is not anywhere near.
* Progress: Similarly, as there is no task in critical section, Task B should not be blocked.

[1819s1] POSIX threads (pthreads) and threads

* Pthread can start on any function as long as the function signature is void\* f(void \*)
* Multi-threaded program using pure user threads can never exploit multi-core processors (as OS is unaware of the threads)
* On a single-core processor that support simultaneous multithreading, we can execute more than one thread simultaneously (SMT)

[1819s1] pseudo-code on giving up CPU before TQ

int main() {

int i, result[100000];

for (i = 0; i < 100000; i++) {

result[i] = compute(i); // compute takes ~1ms per call

if ((i + 1) % 99 == 0)

sleepMillisecond(1);

}

}

The above exploits can be mitigated by keeping track of processor usage across timeslices. Briefly describe with pseudo-code on how you are still able to maximize processor usage unfairly. State all assumptions and explain how the result[] is maintained properly.

int main() {

int i, result[100000];

for (i = 0; i < 100000; i++) {

if ((i + 1) % 99 == 0) {

if (fork() != 0) exit();

}

}

result[i] = compute(i);

}

Assumption: child process do not inherit parent’s usage statistics

Main ideas:

* Spawn child to continue to run. Child is a new process ⇒ highest priority
* Parent should exit to avoid clogging up the system and do redundant work.

Mitigation: forces child process to inherit fully / partly the parent’s CPU statistics

[1920s1] Using many user threads within a process may cause the following problems:

* The chances of stack overflow increase because the threads share the stack space.
* Even with multiple cores, the program will not run faster because only one thread at a time will get to run when the process is scheduled.
* [FALSE] Unix signals cannot be used in programs with multiple threads.

[1920s1] Ubuntu 20.04 (on our lab machines) has elements of microkernel, but is a monolithic kernel.

[2021s1] Assume you are trying to create a shared memory space to be used by two processes. shm\_open is failing because you have too many files open by the process. What can you do to make shm\_open call succeed (choose the least invasive option)? Close a file before calling shm\_open

[2021s1] Assume you are trying to create a process using fork. fork call is failing because you have too many orphan processes in the system. What can you do (choose the least invasive option)?

Run a command in the interpreter to terminate a child process of process init.

* kill is usually shell builtin, so no new process and killing a few processes is definitely less invasive than taking the whole system down.

Reboot the system.

* a command in shell should create a new process and that would not be possible for this system.

[2021s1] In general exec() syscall takes longer time to execute than fork() syscall. Briefly explain why.

fork() duplicates the current process’ image and creates a chile process whereas exec() replaces the current process’ image with a new executable. Duplicating the current process image requires only memory access whereas bringing in a new executable requires access to secondary storage, which is much slower than memory access.

[2021s1] Conditions for a deadlock to occur:

* Only one thread can use a resource at a time (mutual exclusion)
* A resource can be held, then blocking whilst waiting for more resources
* Resources cannot be forcefully taken away from the process holding it
* A circular chain of >= 2 processes, each of which is waiting for a resource held by the next member of the chain

[2021s1] Suppose there are three processes P1, P2 and P3 where P1 accepts some user input and uses message passing to distribute the input to P2 and P3. P2 and P3 should wait for the input from P1, process the input and message the results back to P1. Finally, once P1 has received the results from both P2 and P3, P1 will print the results and exit. Which statements are TRUE regarding the above scenario?

(i) With a blocking receive(), the results from P2 and P3 can be printed in a pre-defined order.

(ii) A non-blocking receive() allows P2 and P3 to work on some other task (if any) while waiting to receive the user input from P1.

[2021s1] synchronisation problems can be avoided using:

Message passing, TestAndSet CPU instructions, Semaphores, Mutexes, Signals

[2021s1] Assume that there is another syscaoll fork\_exec() (fork + exec in one syscall). What is the adv over fork+exec?

The new fork\_exec() incurs less overhead because the memory for the new process is manipulated only once.

[2021s1] CPU&I/O devices scheduling are done separately. Explain the need for different schedulers to be used by the OS.

* Devices have different characteristics/properties; scheduling must be tailored for the characteristics of the device.
* Devices operate independently and can be used simultaneously, thus they need separate queues such that processes can queue for each device separately.
* With independent scheduling and a separate queue, we can account for the usage per device independently by processes. Eg. If a process uses a lot of CPU, that should not affect its IO priority, and vice versa.

[2021s1] Situations when an execution transits from user to kernel mode:

* Syscalls, context switch, interrupt/exception/signal handling
* In hybrid thread model, creating/binding/switching kernel threads

[Tut5] Does disabling&re-enabling interrupts avoid race conditions? Is it a good way?

Yes, equivalent to acquiring a universal lock and releasing it, respectively. No interrupt → no quantum-based process switching (timer interrupts cannot happen). However:

* will not work in a multi-core/multi-processor environment since another process may enter the critical section while running on a different core.
* User code may not have the privileges needed to disable timer interrupts.
* Disabling timer interrupts means that many scheduling algorithms will not work properly.
* Disabling non-timer interrupts means that high-priority interrupts that may not even share any data with the critical section may be missed.
* Many important wakeup signals are provided by interrupt service routines and these would be missed by the running process. A process can easily block on a semaphore and stay blocked indefinitely, because there is nobody to send a wakeup signal.
* If a program disables interrupts and hangs, the entire system will no longer work since it cannot switch tasks and perform anything else.

**Contiguous Memory Management**

1 process occupies 1 partition

Fixed Partitioning

* fixed number of partitions of equal sizeEasy to manage & Fast to allocate
* Internal Fragmentation

Variable-Size Partitioning

* Partition is created based on the actual size of process
* Compaction: Move the occupied partitions around to create bigger, consolidated holes → time-consuming
* First-, Best-, Worst-Fit
* Flexible and removes internal fragmentation
* OS needs to maintain info about the occupied and free memory regions, merge with neighboring holes and split when necessary
* More time to locate appropriate regions

Buddy System

* An array A[0..k], where 2k is the largest allocable size. Each free block is indicated by the starting address.
* 2 blocks are buddy of size 2S if the lowest S bits are identical. e.g. 000000 and 010000 buddy of size 24
* Internal and External Fragmentation both possible
* Allocation and free operations are performed in O(1) (independent of the size of the memory available for allocation)

**Disjointed Memory Schemes**

Paging Scheme

* page size = frame size = 2k where k is any number
* Frame size affected by phy RAM size, not virtual memory
* logical memory is contiguous but physical memory is disjointed
* PTE contains phy frame#, and info bits
* Page table must be contiguous for efficient retrieval of pth entry
* logical page maps to physical frame recorded in page table. Page# as index and frame# as value.
* Physical Address = Frame# \* size\_of(physical frame) + offset
* No external fragmentation
* Insignificant Internal fragmentation → only last page
* Requires 2 memory accesses. 1 to Page table and 1 to actual memory item

Translation Look-Aside Buffer (TLB)

* Hardware support for paging saved in CPU
* Cache for PTEs
* Context switch → TLB entries flushed
* Use page num to search TLB associatively
* TLB-Hit: frame num retrieved to generate phy addr → TLB access time + 1 mem access time
* TLB-Miss: memory access to access the page table → retrieve frame num to generate phy addr → TLB access time + 2 mem access time

E.g. TLB access takes 1ns, Main memory access takes 50ns

Avg memory access time if TLB hit ratio is 90%

= 90% x (1ns + 50ns) + 10% x (1ns + 50ns + 50ns) = 56ns

TLB-miss + page fault:

Direct paging: TLB access time + 1 mem access time + 2 hard disk access time (write in new, write out old)

2-level paging (worst case): TLB access time + 1 mem access time + 4 hard disk access time

Paging Scheme Protection

* Access right bits (WRX)
* Valid bit: Indicates whether the page is valid to access by the process
* every mem access is checked against these bits in hardware
* A process generates a page fault while accessing a memory region for which is does not have access rights.
  + A signal SIGSEGV is sent to the process. The process might terminate.
  + If page found in disk, won't terminate; else dep on OS

Page sharing

* Page table allow several processes to share the same phy mem frame, i.e. same frame num in PTE
* Possible usage
  + Shared code page: same code used by many processes, e.g. libraries
  + Implement Copy-On-Write: parent and child share a page until one tries to modify

Segmentation Scheme

* Logical address space split into segments representing text, data, heap and stack
* mapped into contiguous physical partitions of same size
* Keeps record of segment ID and limit of segment
* LA = <Segment ID, Offset>
* Offset >= limit: segmentation fault
* More efficient bookkeeping
* segments can grow and shrink and be protected/shared independently
* Requires variable-size contiguous memory region → external fragmentation

Segmentation with Paging

* Each segment is composed of pages instead of being a contiguous memory region
* Segment grow by adding a new page then add to its page table

**Virtual Memory Management**

* Logical memory space = Virtual Memory Space = Physical Memory + Secondary Storage (swap: page in SS)
* Swap should not be handled as a normal file:
  + Normal file may be spread across different locations on SS. Paging performance will be affected.
* Only 1 system-wide swap file shared by all processes and per-process swap file is not good
  + OS can preallocate a continuous stretch in secondary storage for the system wide swap file.
  + hard to predict the memory usage of a user program.
* separate address spaces: protect and isolate applications from one another
* the ability to share memory between processes

Extended Paging Scheme

* Memory resident vs Non-memory resident → resident bit to indicate if page is in physical memory
* Page fault: access a non-memory resident

1. Check Page Table
2. If memory resident, access physical location. Done.
3. Else, raise an exception. Page Fault.
4. OS locates page in Secondary Storage
5. OS loads the page into a physical frame.
6. Update Page Table.
7. Return to step 1.

* SS access time >> Physical mem access time
* If mem access often PF → thrashing
* Temporal locality → after page loaded into memory, requested mem location may be needed again soon
* Spatial locality → page contains many consecutive locations that are likely to be accessed in near future

Demand Paging

* Process start with no memory resident page
* Only allocate a page when there is a page fault
* 1st page fault (after making exec()) will be at the new program’s entry point in its text segment where the new program starts executing and fetching instructions (main). This is when the system will load the executable from disk (unless it’s already in use by some other process)
* Fast startup time for new process
* Small memory footprint
* Multiple page fault at the start → slow startup
* Page faults may have cascading effect → thrashing

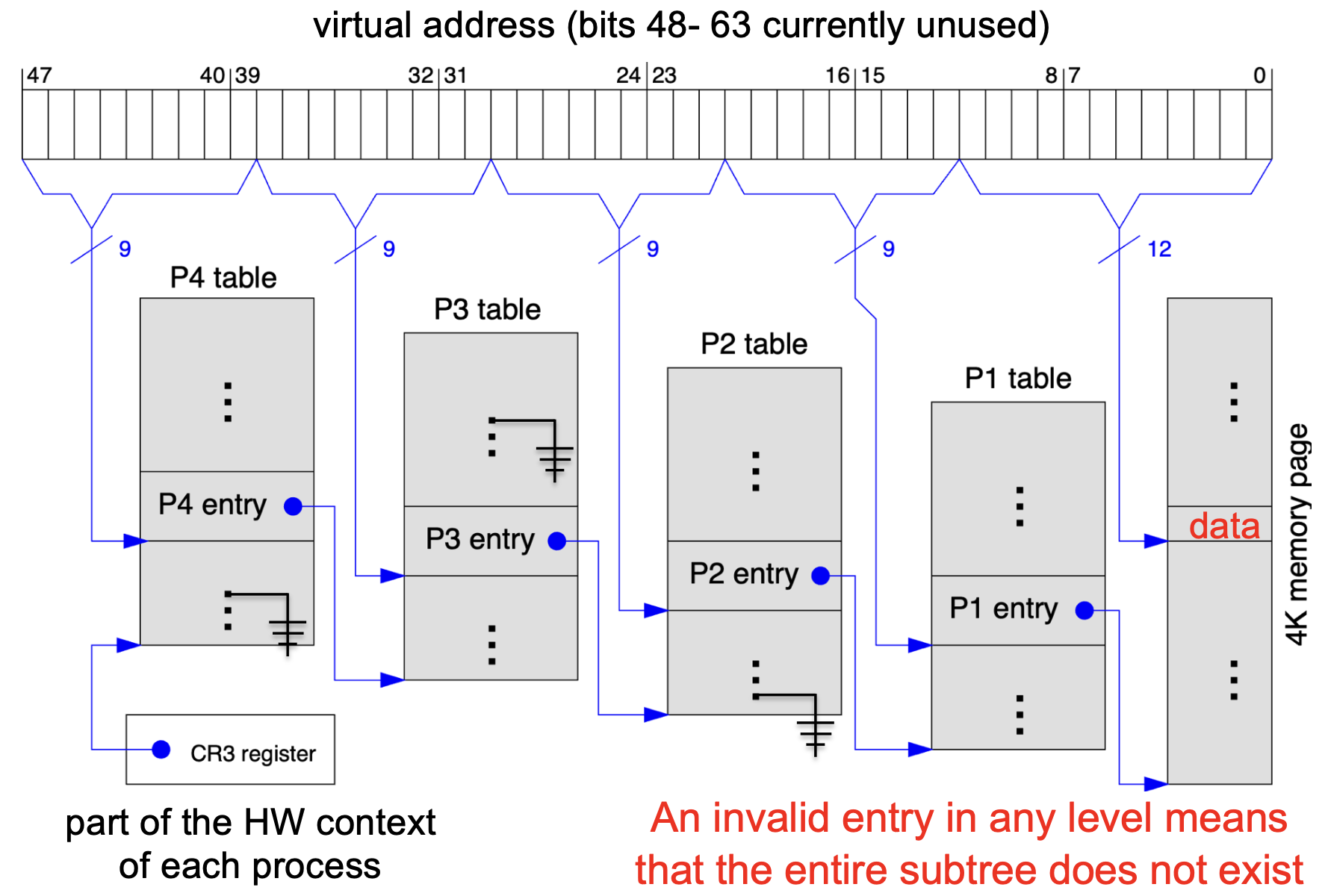
Direct paging

* Keep all entries in a single table

2-Level Paging

* Allocate new region only when needed
* Split page table into smaller page tables, each with page table num
* Each smaller table takes up one page size
* Enables page table structures to grow beyond the size of a frame → page table does not need to be contiguous in OS memory
* Smaller page tables that fit into a page in RAM
* empty entries in the page directory → corresponding page tables need not be allocated
* 2 memory accesses to get frame num
  + TLB eliminates page table accesses but TLB-miss results in longer page-table walks
  + MMU caches: speed up page table walk upon TLB miss

Multilevel paging



* Radix-tree structure
* A table in each level in the hierarchy is sized to fit in a frame
* All tables are setup and wired by the OS in SW, but traversed by HW

Inverted Page Table

* Keep single mapping of physical frame to <pid, page#>.
* Num of PTE = num of frames
* Frame# as index.
* Huge saving: Only 1 table for all processes
* Updating after Page Replacement is much faster
* Slow translation

**Calculation eg**

| * Virtual Memory Address Space = 32 bits * Physical memory = 512MB * Page Size = 4 KB = 212 * PTE Size = 4 bytes |
| --- |
| Direct paging  Num pages = 232 - 212 = 220  Page table size = 220 \* 4 = 4MiB  Total = 4\*4MB = 16MB |
| 2-Level paging  Num PTEs in one smaller page table = 4KB/4 = 210  Num of smaller page tables = num of directory entries =  220 /210 = 210 entries  Page directory table size = 4 \* 210 = 4KB  A process only uses 512MB ⇒ 512MB/4KB = 217pages  Num of smaller page tables used = 217/210 = 27 =128  Overhead for 1 process = 1 page directory + 128 smaller tables = 4KB + 128\*4KB = 516KB  \*note: directory PTEs depends on num of smaller tables, so directory size can be > page size. |
| Inverted table (inverted PTE = 8 bytes)  Num physical frames = 512MB/4KB = 217  A single frame table can be used system wide  Overhead = 217 \* 8 = 1MB |

**Page Replacement Algorithm**

Optimal Page Replacement (OPT)

* Replace page that will not be used for the longest period of time
* Closer to OPT, the better it is
* Guarantees minimum page fault

FIFO

* Memory pages evicted based on their loading time
* Simple to implement. No hardware support needed.
* Belady’s anomaly → more frames, more PF
* Does not exploit temporal locality

Least Recently Used (LRU)

* Temporal locality → Have not used for some time, unlikely to use again
* No Belady’s anomaly, i.e. more frame, less PF
* Substantial hardware support

| * counter for each mem reference in time-of-use field→ replace page with smallest time-of-use value * Need to search through all pages * Possible overflow of t-o-u field | * Use stack. Every time page is referenced, remove from stack and push on top of stack * Replace page at bottom of stack * Hard to implement * Not a pure stack |
| --- | --- |

Second-Chance Page Replacement (CLOCK)

* Modified FIFO
* Each PTE has a reference bit: 1 = Accessed since last reset, 0 = Not accessed

Algo:

1. Oldest FIFO page selected
2. If ref bit == 0 → page is replaced. Done.
3. If ref bit == 1 → Page is given 2nd chance.
   * Ref bit cleared to 0
   * Next FIFO page selected. Go to step 2.

* If all bits == 1, becomes FIFO

Working set

Working set window Δ: an interval of time

W(t, Δ) = active pages in the interval at time t

E.g. count Δ from t (inclusive) forward

| t | … | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| page | … | 2 | 4 | 5 | 3 | 2 | 5 | 2 |

W(9, 3) = {3, 4, 5}

* Δ value is well chosen → the pages in memory are good representation of working set pages for all processes → CPU is well utilized and page fault activity is low
* Δ too small → page fault activity is very high (coupled with low CPU utilization)
* Δ too large → prevent more processes to be runnable to utilize the CPU → both page fault activity and CPU utilization are low
* Pages in memory frames == working set

| Direct page replacement | Inverted table |
| --- | --- |
| To replace frame 2 with p8  1. Search through all process's PTE   * Find one with frame 2 * Update to non-memory resident   2. Use current Process Page Table   * Update page8 to be memory resident and in frame2 | 1. Use Inverted tablet at index 2   * Locate affected PTE, change to non-memory resident   2.Use current process's page table   * Update page 8 to be memory resident and in frame2 |

**File System**

* Physical Memory is volatile and external storage to store persistent information
* Direct access to the storage media is not portable
* Provides:
  + An abstraction on top of physical media
  + High level resource management scheme
  + Protection between processes and users
  + Sharing between processes and users
* Self-contained → info is enough to describe entire org
* Persistent: Beyond lifetime of OS and processes
* Efficient: Good mngmt of free and used space + minimum overhead for bookkeeping

|  | **Memory Mngmt** | **File Mngmt** |
| --- | --- | --- |
| **Underlying Storage** | RAM | Disk |
| **Access Speed** | Constant | Variable disk I/O time |
| **Unit of Addressing** | Address space for process  Implicit when process runs | Non-volatile data  Explicit access |
| **Organization** | Page/Segmentation: determined by HW & OS | Many different FS |

File

* Logical unit of information created by process
* Data: Information structured in some ways
* Metadata aka File attributes:
  + Name, Identifier, Type, Size, Protection, Time, date and Owner, ToC
* File Type:
  + Regular: contains user info
    - ASCII files: txt file, source codes
    - Binary files: executables, predefined internal structure that can be processified by specific program
  + Directories: System Files for FS structure
  + Special Files: Char/block oriented

File Protection

* Controlled access to the info stored in file
* RWX, append, delete, list (read metadata)
* No write → no rename/delete
* Access control list (ACL): a list of user identity and the allowed access type
  + Minimal (same as permission bits)
  + Extended (added named users/grp)
  + Very customisable
  + Additional info
* Permission bits
  1. Owner
  2. Group: users who need similar access
  3. Universe: others
* rwxrw-r-- (owner, group, universe)

Sequential access: start from beginning, cannot skip/rewind

Random access: read in any order

* Read( Offset ): Every read operation explicitly state the position to be accessed
* Seek( Offset ): A special operation is provided to move to a new location in file
* random access is direct access where each record = 1 byte

Direct access:

* Used for file contains fixed-length records,
* Allow random access to any record directly

File operations as syscalls

* Provides protection, concurrent and efficient access
* Maintain info
  + File pointer: current location in file
  + Disk location: actual file location on disk
  + Open count: num of process has this file opened → determine if entry is to be delete in open file table

File Descriptors

File descriptor is an int returned from open(...)

There are 3 standard file descriptors for every program in Unix

* 0 = stdin
* 1 = stdout
* 2 = stderr

These are "opened" automatically and linked to the corresponding files. Note that screen (terminal), keyboard are represented as special files in unix.

* File descriptor table: int array indices that are file descriptors in which elements are pointers to file table entries
  + Saved in PCB

Directory

* Provide logical grouping of files
* Keep track of files

Single-level

Tree-structured

DAG → if a file can be shared

* Hard Link (Files only)

A is owner of file F. B wants to share F.

* Low overhead, only pointers
* Deletion problems.
* Symbolic Link (Files and Directories)
  + B creates special link file G which contains pathname of F
  + When G is accessed, get path of F then access F
* Simple deletion: If B deletes: G deleted, not F, If A deletes: F is deleted, but G remains (not working)
* larger overhead
* Need longer traversal

General Graph → BAD, side effect of Symbolic link

**File System Implementation**

Focus: Hard Disk

Disk Organization: Master Boot Record (MBR) at sector 0 followed by 1 or more partitions. (1 partition = 1 independent FS)

FS contains: OS boot up information, partition details(total number of blocks, number and location of free disk blocks), Directory structure, Files information, Actual file data

File Block Allocation

**Contiguous - directory entry: [file | start | len]**

* Simple to keep track: starting block num + len
* Fast access: only need to seek to 1st block
* External fragmentation
* File size needs to be specified in advance

**Linked list [file | start | end]**

* Each disk block stores the next disk block num (pointer) and actual file data
* File info: 1st and last disk block num
* No fragmentation
* Random access is very slow
* Part of disk block is used for pointer
* Less reliable (1 incorrect pointer renders incorrect file)

**File Allocation Table (FAT, linked list v2.0) [file | start]**

* Move all the block pointers into a single table → in memory
* FAT entry contains either
  + FREE
  + Next block number
  + EOF (NULL ptr)
  + BAD (unusable, i.e. disk error)
* Faster random access: linked list traversal now takes place in memory → 0 hard disk access
* Keep track of ALL disk blocks in a partition → huge memory space

**Indexed Allocation [file | index block]**

* Each file has an index block
* An array of disk block addresses. IndexBlock[N] = Nth Block Address
* Less overhead → only opened files need to be in mem
* Fast direct access
* Limited maximum file size. Max no. of blocks == No. of index block entries
* index block overhead

**Indexed Allocation Variation**

1. Linked scheme - linked list of index blocks
2. Multilevel index - First level index blocks point to a number of second level index blocks
3. Combined scheme - combination of direct indexing and multilevel index scheme (a.k.a. I-Node)
   1. 12 direct pointers → points to disk block
   2. 1 single indirect block → contains no. of direct pointers
   3. 1 double indirect block → points to no. of single indirect blocks
   4. 1 triple indirect block → points to no. of double indirect blocks

**Free Space Management**

Bitmap

* Each disk block rep 1 bit (e.g. 1 = free, 0 = occupied)
* Provide a good set of manipulation: can find the 1st free block, n-consecutive free blocks easily with bit operations
* Keep in memory for efficiency

Linked list of disk blocks

* Each disk block contains:
  + A number of free disk block numbers
  + A pointer to the next free space disk block
* Easy to locate free block: O(1)
* Only the 1st pointer is needed in memory (other ptr can be cached for efficiency)
* High overhead - mitigate by storing the free block list in free disk blocks

**Directory Implementation**

Linear list

* Each entry rep a file
  + Store file name (minimum) and other metadata
  + Store file info or ptr to file info
* Search file takes O(n) - inefficient for large dir and/or deep tree traversal → use cache to rmb recent searches

Hash Table

* Each directory contains a hash table of size N
* Locate a file by filename → filename hashed into index K from 0 to N-1
* Usually chain collision resolution is used i.e. all filenames with same hash value chained together
* Fast lookup
* Hash table has limited size
* depends on good hash function

File information

* Consists of file name and other metadata & disk block info

1. Store everything in dir entry → fixed entry size
2. Store only file name and points to some data structure for other info

**File System Actions**

* Runtime info maintained by OS in memory
* Different processes open same file → different fd returned → different entries in system wide OPT
* A process opens twice the same file → 2 entries added by the process to system wide OPT
* Parent open file → fork → child inherits parent’s file descriptors → both parent and child point to the same entry in system wide OPT

Create (File operation) - To create a file /…/…/parent/F:

1. Use full pathname to locate the parent directory
   1. Search for filename F to avoid dupes. Err.
   2. Search could be cached in directory structure
2. Use free space list to find free disk block
3. Add entry to parent directory

Open /.../…/F:

1. Search system-wide table for existing entry E
   * If found: creates an entry in Process P’s table to point to E and return a pointer to this entry
   * If not found: continue to next step
2. Use full pathname to locate file F
   * If not found → terminate with error
   * When F is located, its file information is loaded into a new entry E in system-wide table → then creates an entry in P's table to point to E n Return a pointer to this entry

Disk I/O Scheduling

Seek - change track

Rotation - change sector

Time taken to perform read/write operation = [seek time] + [Rotational Latency] + [Transfer time]

1. seek time
   1. time taken to position disk head over the proper track [2ms - 10ms]
   2. total seek time/total # of possible seeks → ⅓N, where N is the time for max seek distance
2. Rotational Latency
   1. time taken for desired sector to rotate under the read/write head
   2. 4800 - 15000 rpm → [12.5ms - 4ms per rotation]
3. Transfer Time
   1. Time taken to transfer the sectors
   2. Transfer Time = Size/Rate

(1)+(2) >> (3)

* Scheduling to reduce seek time (diff to mitigate rotational latency)

FCFS

Shortest Seek First (SSF)

* Seek time for each track to be given

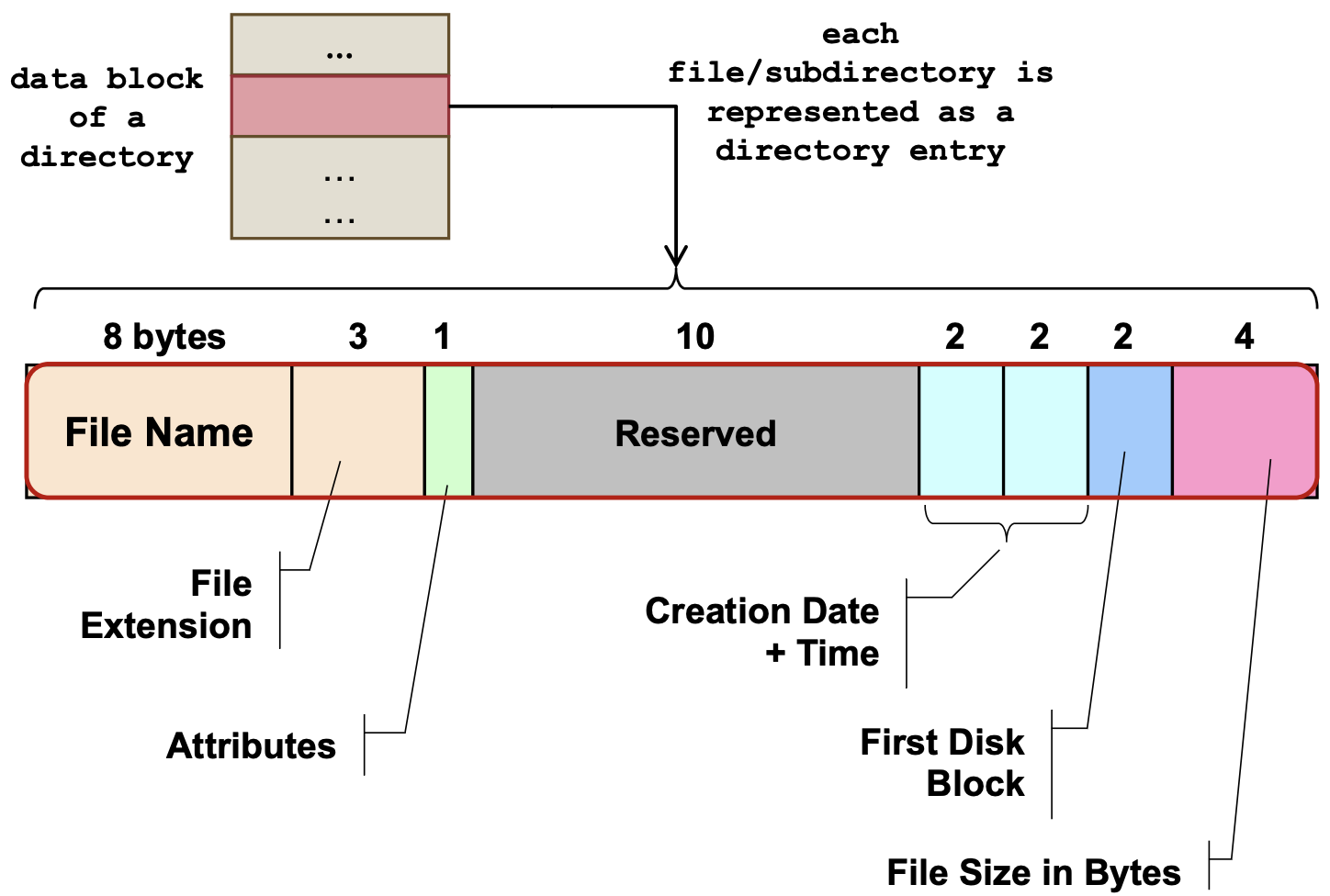
SCAN (elevator)

* Bidirectional (innermost <-> outermost)
* 1-directional (outermost → innermost): C-SCAN

**Microsoft FAT**

FAT08: 8 bits to rep disk block index

Directory entry:



1. Use first disk block number stored in directory entry to find the starting point of the linked disk blocks
2. Use FAT to find out the subsequent disk blocks number → terminated by special value (EOF)
3. Use disk block number to perform actual disk access on the data blocks

File deletion

* [Dir entry] set 1st letter in filename to **0xE5**
* [FAT entries] set FAT entries in linked list to FREE
* [Disk blocks] Actual data blocks remain intact - can attempt to undelete
* DO NOT keep track of free space → must be calculated by going through the FAT

Disk cluster

* Contiguous disk blocks
* Page size = same or multiple of cluster size
  + For pages to be efficiently swapped out
* Larger cluster size → larger usable partition
* Larger cluster size → larger internal fragmentation

FAT size

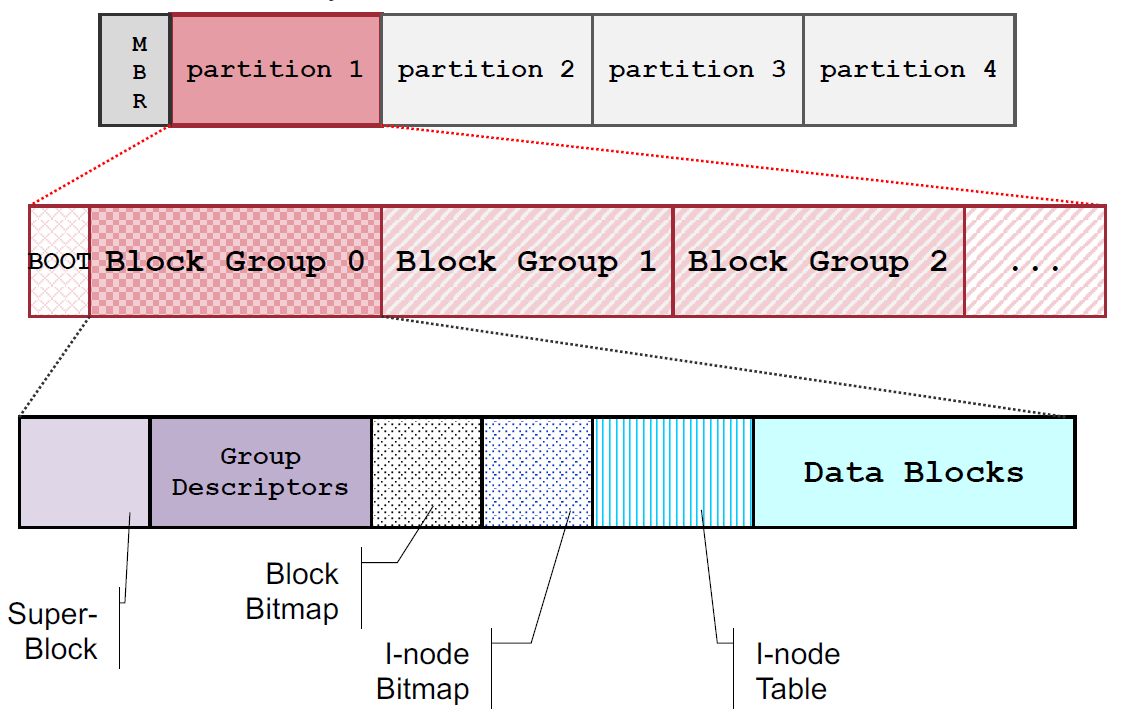
* Bigger FAT → More disk block/cluster → More bits to represent each disk block/cluster
* Actual size is a lil less - special values (EOF, FREE)

E.g. 4KiB cluster

| FAT12 | FAT16 | FAT32 |
| --- | --- | --- |
| 212 clusters  Largest partition: 4KB\*212=16MB | 216 clusters  Largest partition: 4KB\*216=256MB | 228 clusters  Largest partition: 4KB\*228=1TB |

**Extended-2 FS**

* Disk space split into blocks that correspond to 1 or more disk sectors
* Blocks are grouped into Block Groups
* I-Node == Each File/Directory
  + Contains: File metadata + data block addresses



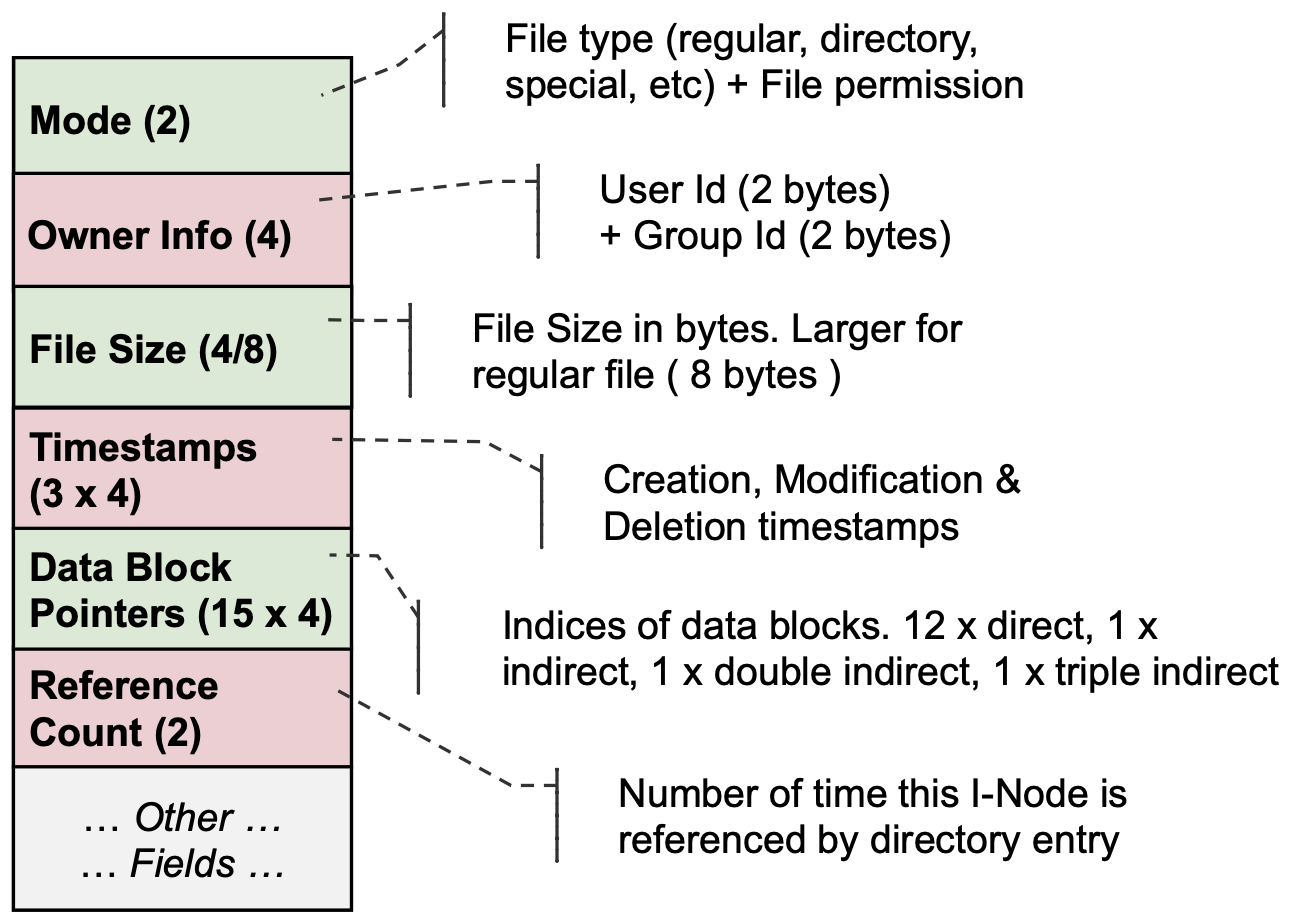
* Superblock: describes whole FS e.g. total I-Node #, I-Node per group, total disk blocks, disk blocks per group
* Group Descriptors: Describe each of the block group, duplicated in each Blk grp
* Block Bitmap: keep track of usage status of blocks in grp
* I-Node bitmap: keep track of status of I-Nodes in this BG
* I-Node table: array of I-Nodes of I-Nodes in this BG
* data blocks for a file come from the same block group to reduce fragmentation

E.g. Each disk block address is 4 bytes, Each disk block is 1KiB

* indirect block can store 1KiB/4 = 256 addresses
* Maximum File Size = Direct blocks + single indirect + double indirect + triple indirect

= 12x1KiB+256x1KiB+2562 x1KiB+2563 x1KiB = 16843020 KiB (16 GiB)

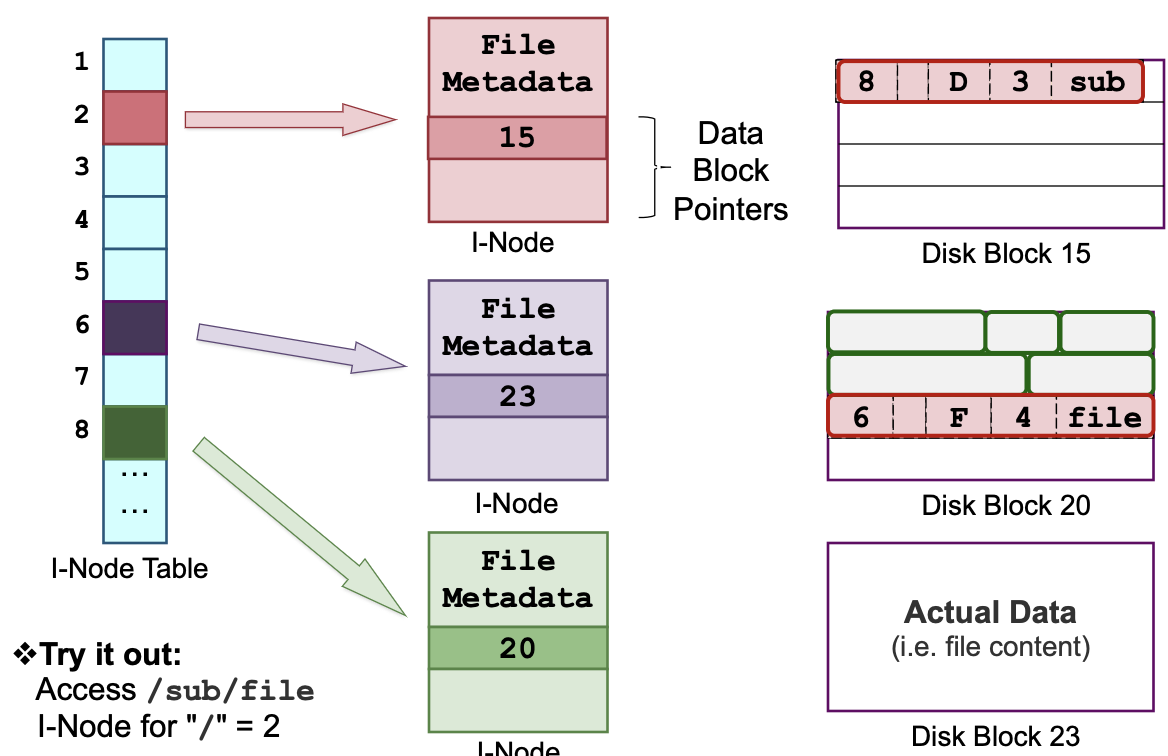
* Each directory entry contains I-node num, entry size, type (file/subdir), length of file/subdirectory name, file/subdir name (up to 255 char)
* Access time to a file depends on the on X (num files in direct blocks) and the number of indirect folders. The access time for a file is X (logXN + 1)



Locate file in Ext2

e.g. /sub/file, curr dir = root (inode num = 2)

1. Read i-node 2 from i-node table → get data block ptr from i-node → access disk block
2. If it is a directory, e.g. sub/, locate directory entry in currDir (disk block) → get i-node num → read i-node to get next disk block ptr to file
   1. currDir = next part in path name
   2. Go to step 2
3. If it is a file → locate dir entry in currDir → get i-node → read i-node for block ptr → get actual content



Deletion in Ext2

1. Remove its directory entry from the parent directory
   1. Point prev entry to next entry
   2. If first entry → blank record
2. Update I-Node bitmap:
   1. Mark corresponding I-Node as free
3. Update Block bitmap:
   1. Mark corresponding block as free

* Hard/Symbolic Link with I-Node
* Directory A contains file x with I-Node# xn.
* Directory B wants to share x.
* Hard Link: creates a directory entry in B which uses the same I-Node number xn but can have different filename.
* Symbolic link: creates a new file Y in B. Y contains the pathname of X.

[1617s1] Abstraction and protection for memory and file management

| **Memory** | **File** |
| --- | --- |
| illusion that process owns the entire memory space. | Files is a single contiguous logical entity. |
| Memory space of each process is mapped to different phy addr, isolating them from each other. | Files can only be opened through system call, OS can prevent files from being opened for incompatible operations. |

[1617s2] Why do most OS utilize a bootstrap sequence to load and start?

It allows the OS to first start and run a simple portion of itself, and as more capabilities are loaded and more of the system is initialized, the bootstrap sequence allows the OS to load and start more and more sophisticated chunks of itself.

* E.g. bootstrap routine in boot sector loads up core disk services, and use this to load up memory and process management routines, and use that to load up services like web servers, ssh servers, etc, and finally load up the shell.

[1718s1] Regions the content resides in

* Page table: OS memory (PCB stores ptr to page table)
* (system-wide) Open file table: OS memory (not PCB)
* Dynamically allocated data structures & fd from open(): user process pages in phy RAM, SWAP (non-mem resident user process pages), virtual memory space
* Code for process scheduler/page fault/fork handler: OS
* Binary of a compiled prog, e.g. a.out: not in virtual mem

[1718s1] program to find out page size

1. Create an arr of large size
2. Access item sequentially and monitor the time needed for each access → huge spike → page boundary hit

[1718s1] resources to be cleaned up by OS after exit

* Mem frames (use page table to find out)
* All opened files (use fds to find out)

[1718s1] Suppose the hard disk has only 4 disk blocks (sectors) per track. If a file takes up 8 disk blocks, does it incur additional rotational delay once we locate the 1st disk block?

* Yes. It occupies 2 tracks, there may be rotational delay between the change of track.
* Unless this was taken into account during the allocation (i.e. the disk block on the second track is chosen such that the rotational delay is canceled by seek delay. This idea is known as disk skew)

[1819s1] Is it possible that the execution of a simple "memory load" instruction causes the currently running process to enter block state?

Yes, when the memory load causes a page fault. Swap pages need to be brought in, i.e. disk I/O.

[1920s1]Is it possible that the execution of a file read operation (on an opened file) does not cause the currently running process to enter block state?

Yes, OS / Library can have in-memory buffer for file content to provide. So, the file operation actually just read from the buffer instead of the file, i.e. no disk I/O.

[1819s1] largest num of disk accesses to file "/WHY/FAT08" with some disk block in single indirect = 7 = 1 ( "/" inode) + 1("/" DEs) + 1( "WHY/" inode) + 1 ("WHY/" DEs) + 1 ("FAT08" inode) + 1 (single indirect) + 1 (file content)

[1920s1] A program is running for the first time in an operating system and it is stopped. Later, the same program (process) is started again. We observe that the startup time the second time is much shorter than the first time. The most important source of overhead in starting for the first time is:

1st time: the executable file is first brought from disk to RAM;

2nd time: exe is alr in RAM

[1920s1] A program is processing large amount of data in memory. After all data has been read within the memory space of the process, the program appears to make much slower progress than expected, although there are no out-of-memory errors. You are running on a system with one core, but the CPU load is near zero. A suggestion is to increase the SWAP space size to better accommodate the program’s virtual memory requirements. Will this suggestion help to improve the observed phenomenon?

* large memory consumption, low CPU utilization, slow progress → thrashing – a state where processes spend most of their time paging data in and out from/to disk.
* The root cause is a shortage of physical memory.
* adding swap space will not solve the problem. In fact, if a shortage of swap space had been an issue, the program would have seen out of memory errors (malloc() failing, or in Linux the OOM killer killing the program.)

[1920s1]

A process is opening a file and creates multiple threads reading from the same file (file descriptor) using system call read. The threads will read the same content. FALSE

* Read moves the ptr (pts to same FD) → fd changes per read → diff offset

A process creates multiple threads, each thread opens the same file and reads 32 bytes from the file (using read system call). Each thread reads the same info. TRUE

* Each thread open same file → multiple fds returned

A process creates multiple process, each process opens the same file and reads from the file (using read). Each process read same information. TRUE

A process is opening a file and continues to execute its code. Later, the process closes the file and finishes its execution, but the entry in the open file table is not freed. Explain why.

The process had a fork after opening the file.

[2021s1]

A computer supports 64-bit virtual memory addresses. The page size is 1KiB (210bytes), PTE size = 2 bytes.

Assuming the addressing is at the word (8 bytes) level, calculate the size of the standard page table (in bytes).

Since the addressing is at word (8 bytes) level, OS supports 264\*8 bytes ⇒ (264\*23\*21)/210 = 258

A process has 3 threads, T1, T2, and T3. T1 opens 3 files and T2 creates a pipe. After these actions, T1 executes a fork() command and T2 executes a fork() whose child immediately calls execvp() to execute a single-threaded program. T1 closes the three files and then terminates. T3 terminates without opening any file. How many file descriptors have been used (opened) by this program?

* The initial process has 3 fds (stdin, stdout, stderr).
* T1 creates 3 fds, T2 creates 2 fds (pipe) ⇒ 8 now
* After this, there are 2 fork() calls → each process will duplicate the fd table with 8 fds: 24 fds are created in total.

Implement fork\_file() to reduce duplicate data content

int fork\_file(const char \*file\_path, const char\* new\_file\_path)

* fork\_file() takes in a file path and creates a new directory entry in the path of the new\_file\_path duplicates the inode of file\_path keeping the same data block pointers,
* but new timestamps (creation/modification set to current time) and reference count (set to 1).
* returns 0 on success, or -1 if unsuccessful.
* Similar to hard link but inode duplicated
* When the "fork" (new\_file\_path) is modified or truncated, the source file will not change, and new disk blocks will be allocated when necessary. (copy-on-write)
* Basically, we need to know which disk blocks are shared among multiple files. This info needs to be part of the group (partition) information in EXT2
  + Alternative: a reference count for each disk block. Whenever a file is being forked, the disk blocks in that file should have their reference counts incremented.
  + each data block needs to have an associated permission bit. Initially for a writable file, the permission bit for all blocks is 1. However, after a fork\_file, all data blocks of the forked file and new file have permission bits set to 0. If a write is called on a writable file, at a file offset that points to a data block that has permission bit 0 that block will be copied to a new block and the copy will have permission bit set to 1.

[Quiz 10] Processes P and Q are running on a system that supports virtual memory. Process P executes an instruction that causes a page fault, after which the operating system evicts one page of process Q from memory to create a free frame for P's new page.

* The OS performed global page replacement.
* System X uses demand paging.

int main{

int fd1 = open("/path/to/file/A", O\_RDONLY);

fork();

int fd2 = open("/path/to/file/B", O\_RDONLY);

// point of interest

return 0;

}

Assume that the parent and the child are both currently executing the instruction at the point of interest.

* At the point of interest, there will be exactly 1 entry for file A in the system-wide table of open files.
* At the point of interest, both parent and the child will have a separate an entry for A in their per-process table of open files.
* At the point of interest, there will be exactly 2 entres for file B in the system-wide table of open files.

[Quiz 12]

Hierarchical page-tables are a particularly good example of which system design principle(s)? INDIRECTION

In modern Intel/AMD/ARM cores, there is a separate TLB for instruction accesses (iTLB) and data accesses (dTLB). Why

TLBs have extremely tight latency requirements: every instruction and data accesses it

1. Instruction and data accesses often exhibit different temporal and spatial locality patterns

* E.g. instruction accesses are much more sequential compared to data accesses.
* The TLBs are separated so that each of them can be specialized to serve its purpose faster and with a more suitable replacement policy.

2. two smaller TLBs are faster than 1 bigger. (same for other hardware structures)

3. allows better sizing of each TLB. dTLB is usually larger than iTLB, because the amount of memory used for code is typically much smaller than the memory used for data, and because instructions have much higher locality, both spatial and temporal.

4. Mixing PTEs of data and instructions in the same TLB would allow data PTEs to kick out important instruction PTEs, because data is usually much larger. Separating the 2 TLBs will isolate instructions from data and guarantee a high TLB hit rate for instructions.

What are the effects of using a larger page size?

* Decreased TLB pressure (less TLB entries required): each entry covers bigger memory space
* Increased internal fragmentation
* Decreased space overhead of page tables: fewer PTEs
* Decreased latency of page-table walks: fewer PTEs → fewer levels in page-table hierarchy