2. Process Management

Life of a C program

Preprocessor

The preprocessor expands directives such as #include , #define , #ifdef ...

Compiler

The compiler takes the expanded C code, checks the syntax, and generates... Assembly code (gcc); LLVM IR (clang).

In the meantime, it also optimizes the code. (gcc -S -O1/O2/O3 program.c)

Assembler

The assembler converts the generated assembly code to an object file.

The object file contains machine code, but isn't yet executable.

Linker

The linker combines object files and libraries to produce the executable file.

A library is just a collection of functions and variables.

Static & dynamic linking: the final program embeds all library functions vs. load dynamic libraries when the program runs.

Processes

The process is the most central concept in an operating system.

- It's an abstraction of a running program.
- It attaches to all the memory that is allocated for the process.
- It associates with all the files opened by the process.
- It contains accounting information such as its owner, running time, memory usage...

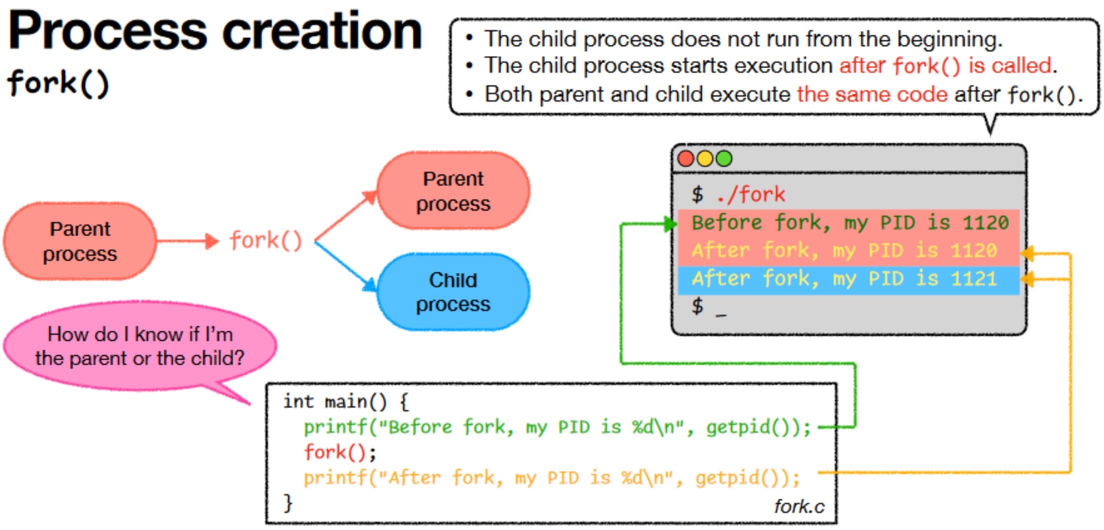
Let's start with some system calls.

getpid() and getppid()

The OS gives each process a unique identification number, the Process ID (PID).

- getpid() returns the PID of the calling process.
- getppid() returns the PID of the parent of the calling process.

fork()



parent process → parent process + child process, executing the same code

To know in which process — fork()'s return value differs for the parent and the child:

- In the parent, fork() returns the PID of the child process > 0
- In the child, fork() returns 0

The child inherits the parent's... Program code; Memory. Differs from the parent in… Return value of fork(); Process ID; Parent; Running time.

fork buffer: printf() invokes the write() system call; buffer combines several printf() for one write() system call, to reduce the number of system calls; at fork(), the child inherits the buffer.

Unbuffered: invoke write() immediately. Line-buffered: invoke write() when a newline character. Fully-buffered: invoke write() when the buffer becomes full or before the process terminates.

On a single-processor system, only one process can be executed at one time. After fork(), does the parent or the child run first? — We don't know. Decided by process scheduling.

execve() and the exec*() family of functions

When execve() is called, the process replaces the code that it's executing with the new program and never returns to the original code — nothing after execve() will be executed.

execl, execlp, execle, execv, execvp, execve.

Path name or file name

- Default — path name, e.g., "/bin/ls".
- p — file name, e.g., "ls".

Argument list or array

- l — list, e.g., execl("/bin/ls", "/bin/ls", "-a", "-l", NULL);
- v — array, e.g., execv("/bin/ls", argv);

Environment variables

- Default — inherit the current environment.

- e — specify a new environment array.

Environment variables are a set of strings maintained by the shell.

PATH=/user/bin; SHELL=/bin/bash; USER=cj2133; HOME=…; PWD=…

Process creation and execution & system()

Can we just run a program and be able to come back?

Yes, there is a convenient library function (not a system call): system()

It is implemented using fork() and exec() to call /bin/sh -c command

```
int my_system(const char *command) {
    if (fork() == 0) {   // in child
        execl("/bin/sh", "/bin/sh", "-c", command, NULL);
        exit(-1);        // exits if execl() fails, returns -1 to parent
    }
    wait(NULL);          // waits until child terminates
    return 0;
}
```

wait() suspends execution of the calling process until one of its children terminates.

waitpid() waits for a particular child; waits for a stopped/resumed child.

Summary

A process is created by cloning.

- fork() is the system call that clones processes.
- Cloning is copying — the new process inherits many things but not all.

Program execution is not trivial.

- A process is the entity that hosts a program and runs it.
- A process can run more than one program.
- The exec*() system call family replaces the program that a process is running.

Processes in the kernel

Kernel-space memory vs. user-space memory

Processes in the kernel — Task list (”doubly-linked list”), “node” — process control block, contains everything related to a process.

How to redirect stdout to a file? — File descriptors, print to the file instead of the screen.

Process execution

Going back and forth...

A process switches its execution from user mode to kernel mode by invoking system calls.

When the system call finishes, the execution switches from kernel mode back to user mode.

Handling system calls

Example: the getpid() system call

The CPU is running a process in user mode.

It wants to invoke the getpid() system call. Each system call has a unique syscall number.

The process puts the syscall number of getpid() in a specific CPU register.

Then it executes a TRAP instruction to switch from user mode to kernel mode.

The kernel starts execution at the syscall dispatcher. It examines the syscall number, looks up the syscall table, and invokes the corresponding syscall handler.

The sys_getpid() handler reads the Process ID of the calling process from task_struct.

Then it executes a RETURN-FROM-TRAP instruction to switch from kernel mode to user mode.

Execution of a process: user time vs. system time

Process-management syscalls in the kernel

fork()

execve()

wait() & waitpid() & exit() → child enters the zombie state

Signals

Signals are software interrupts. It's a form of inter-process communication (IPC).

A process can send a signal to another process.

When a signal arrives, the OS interrupts the target process's normal control flow and executes the signal handler.

How are signals generated?

From the user space (keyboard, commands, kill()…) / the kernel or hardware (SIGCHLD…)

The kill command sends SIGTERM to the target process by default. The default signal handler of SIGTERM is to terminate the process.

Foreground and background jobs run concurrently. fg: resumes a job in the foreground (send a SIGCOUT signal and then wait for the child).

Just like the kill command, the kill() syscall sends a signal to a process.

The raise() library function sends a signal to itself (by calling kill() with its pid).

The signal() system call changes the signal handler.

The pause() system call puts the process to block (sleep) until the delivery of a signal handled by the process, or a signal terminating the process.

Process organization

Booting the computer

BIOS (Basic input/output system) is firmware (i.e., software providing low-level control for hardware). Locates the boot device → boot code determines the bootable partition → executes boot loader → starts the OS kernel → creates the first process (init/systemd)

Orphans: If the shell terminates, the program becomes an orphan. The init process automatically adopts all orphaned processes (becomes their new parent).

Processes in Linux are organized as a single tree (instead of a forest).

Reparenting allows processes to run without a parent shell.

Process scheduling

Multiprogramming: Computers often do several things concurrently, even if it has only one CPU.

Multitasking: The CPU switches from process to process quickly, running each for a few ms.

Scheduling : The OS needs to choose which process to run next. The part of the OS that makes the choice is called the scheduler.

Process states

Ready: runnable but temporarily stopped to let another process run.

Running: the process is actually using the CPU at that instant.

Blocked: unable to run until some external event happens. interruptible/uninterruptible.

Zombie: the process exits and its parent has not yet waited for it.

The context of a process consists of its user-space memory; register values.

The scheduler decides which process to run next. If the next process is different from the current one, the OS performs a context switch (saving and restoring registers; switching memory maps…).

Most processes' execution alternates between CPU execution and I/O wait.

CPU-bound processes spend most of their running time on the CPU. user time > sys time.

I/O-bound processes spend most of their running time on I/O. user time < sys time.

Nonpreemptive scheduling: a process keeps running until... it starts waiting for I/O; or it voluntarily relinquishes the CPU.

Preemptive scheduling: a process can run for a particular period of time, then suspended and another process may run.

What's a good scheduling — general goals

- Fairness: each process should have a fair share of the CPU.

- Policy enforcement: the system's policies should be carried out.
- Balance: all parts of the system should be kept busy all the time.

Batch systems: process jobs in batches, goals: high throughput, short turnaround time

Interactive systems: have interactive users, preemption is essential, goal: short response time

Real-time systems: guarantee response within specified time constraints, goal: meet deadlines

Scheduling algorithms

Metrics

Wait time — the duration that the job is in the system but not running

Turnaround time — the duration from when the job arrives in the system to it completes. $T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$

Response time (in interactive systems) — the duration from when the job arrives in the system to the first time it is scheduled. $T_{\text{response}} = T_{\text{first run}} - T_{\text{arrival}}$

First-come, first-served (FCFS), a.k.a. first-in, first-out (FIFO), is a nonpreemptive scheduling algorithm in batch systems.

Shortest job first (SJF) has both nonpreemptive and preemptive (shortest remaining time) versions.

Round-robin (RR) is a preemptive scheduling algorithm in interactive systems.

Each job is assigned a time slice (a.k.a. quantum), the amount of time the job is allowed to run. Jobs are running one by one in the run queue.

RR has worse CPU efficiency than SJF. However, jobs on a RR scheduler are more responsive. This is an inherent trade-off between performance (SJF) and fairness (RR). RR needs more context switching, which is relatively slow.

Priority scheduling: Each job is assigned a priority. The scheduler always chooses the job with the highest priority to run. Priorities can be static or dynamic.

Static priority scheduling limitations: low-priority jobs may starve to death.

Dynamic priority scheduling: no standard way.

Multilevel feedback queue (MLFQ): It's a kind of dynamic priority scheduling, but each priority has its own policy. MLFQ observes how jobs behave over time, and prioritize them accordingly.

All jobs start running at the highest priority. When a job uses up its time slice, its priority is reduced by 1. If a job gives up the CPU before the time slice is up, it stays at the same priority.

Limitation if a program changes its behavior over time → new rule: after some time period, move all jobs to the highest priority (priority boost).

Interprocess communication

Processes often need to communicate with one another.

IPC: to share information, to reuse software, to speedup computation (e.g. MapReduce).

Case study: the pipe() system call returns two file descriptors: pipefd[0] and pipefd[1]. This is called a producer-consumer model.

The kernel provides a form of synchronization. However, for shared memory, it's up to the processes to coordinate.

Race condition: race to access the shared resource. The results depend on the timing of the execution, i.e., the particular order in which the shared resource is accessed.

To avoid race conditions, we need mutual exclusion: If one process is accessing a shared resource, the other processes must be excluded.

A critical section (a.k.a. critical region) is a piece of code that accesses a shared resource. It

should be as tight as possible; it may access multiple shared resources.

Requirements:

- No two processes may be simultaneously inside their critical sections — mutual exclusion
- No assumptions may be made about the speeds or the number of CPUs
- No process outside critical section may block others → all process can make progress
- No process should have to wait forever to enter its critical section → bounded waiting

Spinlocks are built upon CPU instructions that guarantee atomic operation.

A semaphore is an object with a non-negative integer value.

The value must be initialized before being used.

There are two operations: down() and up()

```
void down(semaphore *sem) {
    // atomic operation
    if (*sem > 0) {
        *sem = *sem - 1;
    } else {
        block on sem;
    }
}

void up(semaphore *sem) {
    // atomic operation
    if (some process is blocked on sem) {
        let one such process proceed
    } else {
        *sem = *sem + 1;
    }
}
```

Classical IPC problems

The producer-consumer problem

- It models access to a bounded buffer.

The dining philosopher problem

- It models processes competing for exclusive access to a limited number of resources (e.g., I/O devices).

The readers and writers problem

- It models access to a database.

The sleeping barber problem

- It models a queueing situation

The producer-consumer problem (the bounded-buffer problem)

Two processes share a fixed-size buffer.

The producer inserts data to the tail of the buffer.

The consumer removes data from the head of the buffer.

Requirements:

1. When the producer wants to insert an item into the buffer, but the buffer is already full...

- The producer should block.
- The consumer should wake up the producer after it has consumed an item.

2. When the consumer wants to remove an item from the buffer, but the buffer is empty...

- The consumer should block.
- The producer should wake up the consumer after it has produced an item.

3. Mutual exclusion: no two processes can access the shared buffer at the same time.

Implementation using semaphores

```
semaphore mutex = 1;           // controls access to critical section
semaphore empty = BUFFER_SIZE; // counts empty buffer slots
semaphore filled = 0;          // counts filled buffer slots
```

Why do we need three semaphores?

```
void producer() {
  int item;

  for (;;) {
    item = produce_item();
    down(&empty);   // decrement empty count
    down(&mutex);   // enter critical section
    insert(item);   // put new item in buffer
    up(&mutex);     // exit critical section
    up(&filled);    // increment filled count
  }
}
```

```
void consumer() {
  int item;

  for (;;) {
    down(&filled);    // decrement filled count
    down(&mutex);     // enter critical section
    item = remove();  // take item from buffer
    up(&mutex);       // exit critical section
    up(&empty);       // increment empty count
    consume_item(item);
  }
}
```

mutex guarantees mutual exclusion.

empty and filled are for synchronization.

empty represents the number of empty slots.

filled represents the number of filled slots.

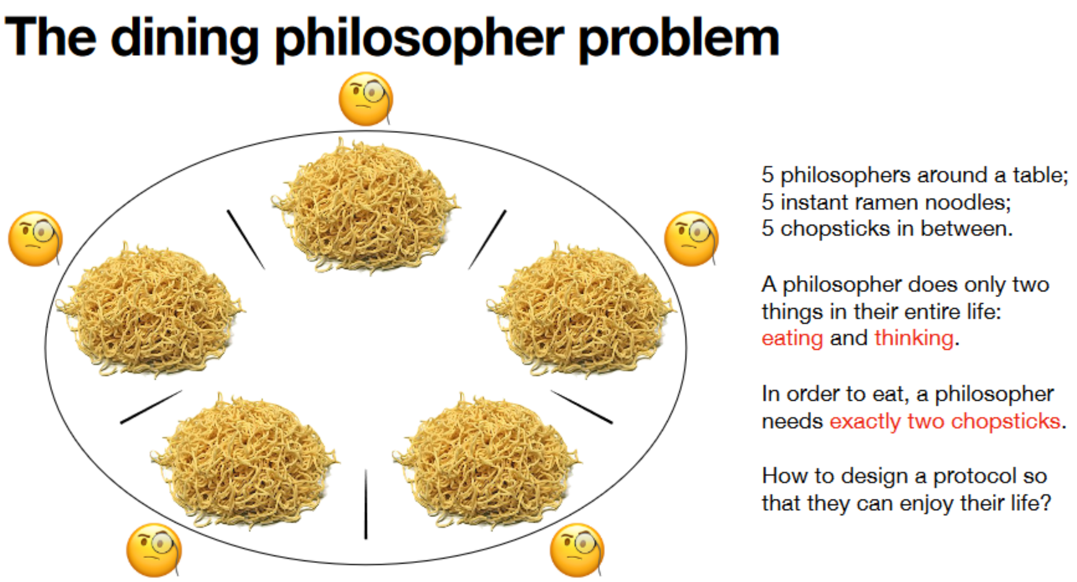
Invariant: empty + filled = buffer size

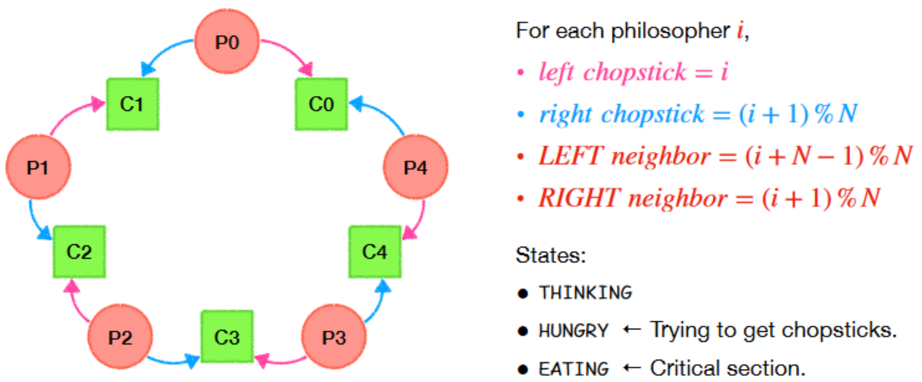
If we swapped lines 6 & 7 in the producer, the scenario is called a deadlock.

It happens when there is a circular wait...

- The producer is waiting for the consumer to up() the empty semaphore.
- The consumer is waiting for the producer to up() the mutex semaphore.

The dining philosopher problem





Requirements:

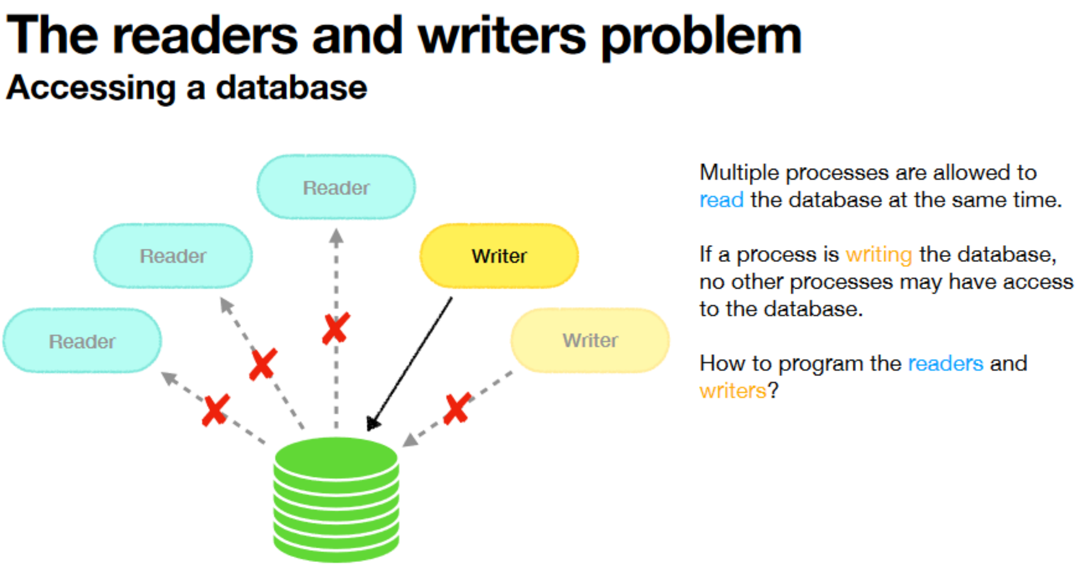
1. Mutual exclusion
2. Synchronization: the solution should avoid any potential deadlock or starvation scenarios.

Potential issues:

- Modeling each chopstick as a semaphore: we are afraid to down() a chopstick, as that may lead to a deadlock.
- However, our real issue is that philosophers need to eat, not chopsticks. We should guarantee: when a philosopher is eating, their left and right neighbors cannot eat.

```
int state[N];             // THINKING, HUNGRY, or EATING
semaphore mutex = 1;
semaphore sem[N];         // what are the initial values?

void take_chopsticks(int i) {
  down(&mutex);
  state[i] = HUNGRY;
  test(i);          // try to take both chopsticks
  up(&mutex);
  down(&sem[i]);  // block if cannot take chopsticks
}

void put_chopsticks(int i) {
  down(&mutex);
  state[i] = THINKING;
  test(LEFT);    // see if left neighbor can now eat
  test(RIGHT);  // see if right neighbor can now eat
  up(&mutex);
}

void test(i) {
  if (state[i] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING) {
    state[i] = EATING;
    up(&sem[i]);  // take both chopsticks
  }
}

void philosopher(int i) {
  for (;;) {
    think();
    take_chopsticks(i);  // section entry
      // block until I take both chopsticks
    eat();               // critical section
    put_chopsticks(i);   // section exit
  }
}
```

The readers and writers problem



Requirements:

1. Mutual exclusion: the database is a shared resource.
2. Synchronization

- When a reader is reading, other readers are allowed to read the database.
- When a reader is reading, no writers are allowed to write the database.
- When a writer is writing, no readers or writers are allowed to access the database.

3. Concurrency: concurrent access from multiple readers should be allowed.

```
semaphore db = 1;        // controls access to the database
semaphore mutex = 1;     // controls access to "reader_count"
int reader_count = 0;    // # of processes reading or wanting to read
```

```
void reader() {
  for (;;) {
    down(&mutex);
    if (++reader_count == 1)
      down(&db);   // the first reader locks db
    up(&mutex);
    data = read_database();  // critical section
    down(&mutex);
    if (--reader_count == 0)
      up(&db);     // the last reader unlocks db
    up(&mutex);
    consume_data(data);
  }
}
```

```
void writer() {
  for (;;) {
    data = produce_data();
    down(&db);              // locks db
    write_database(data);   // critical section
    up(&db);                // unlocks db
  }
}
```

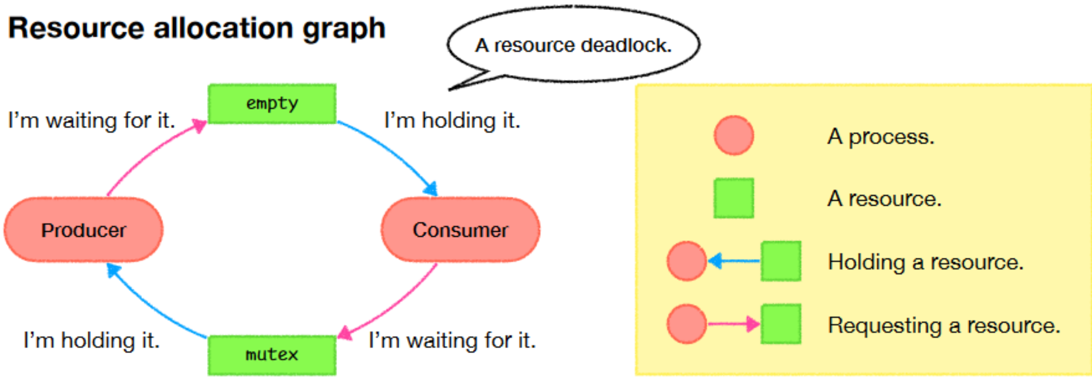
This solution meets all our requirements.

However, **it gives readers a higher priority than writers**.

As long as there is a steady supply of readers, writers will suffer from starvation.

Deadlocks

Remember the deadlock in the producer-consumer problem?



Conditions for resource deadlocks:

Mutual exclusion

- Each resource is either available or currently assigned to exactly one process.

Hold and wait

- Processes currently holding resources that were granted earlier can request new resources.

No preemption

- Resources previously granted cannot be forcibly taken away from a process.
- They must be explicitly released by the process holding them.

Circular wait

- A cycle of 2+ processes, each of which is waiting for a resource held by the next member of the cycle.

Preventing deadlocks

The most adopted approach is the ostrich algorithm — stick your head in the sand and pretend there is no problem.

Attack #1: no mutual exclusion?

Method #1: make data read only; so processes can use the data concurrently.

Method #2: implement lock-free data structures.

- test-and-set; compare-and-swap (CAS)

Method #3: read-copy-update (RCU).

- It allows a writer to update the data structure while other processes are still using it.
- A reader would see and traverse either the old version or the new version.

Attack #2: no hold-and-wait?

Method #1: request all resources at once before starting execution.

- Drawback: a process may not know what resources they will need; resources held for longer than needed, thereby decreasing concurrency.

Method #2: release all resources before requesting a new one.

Attack #3: no "no preemption"?

Technically, we can't forcibly take away a resource that a process already holds. In practice, a process can "preempt" their own ownership in a graceful way. We used this approach in the diningphilosopher problem. It's called trylock.

Attack #4: no circular wait?

It's probably the most practical and frequently used approach.

All the resources are given a total order (i.e., global numbering). A process can request only resources higher than what it's already holding.

Detection and recovery

Allow deadlocks to occur (occasionally); Try to detect when this happens; take action to recover

Detecting deadlocks

A deadlock detector runs periodically and build the resource allocation graph; find a cycle.

Threads: lightweight processes

A thread is an execution entity within a process.

A multithreaded process can have more than one execution in it

- All threads share the same code.
- A new thread starts with a specific thread function.
- The thread function can invoke other functions and system calls.
- However, the thread function does not return to its caller.

All threads share the same global variables and dynamically allocated memory.

Each thread has its own stack for local variables. We can still access another thread's stack if we know the memory address.

It allows multitasking within a process:

- One thread waits for the user input;
- Another thread performs computation.

$\longrightarrow$ Better performance and responsiveness.

Threads are easier to create and destroy than processes (10-100× faster).

Threads share the same address space; so sharing data is easy.

Thread models

Many-to-one model

- Implement threads in user space.
- Cons: When a blocking system call is invoked, all threads will be blocked; Page faults (discussed later) in a thread will block the entire process; No preemption of threads due to the absence of clock interrupts.

One-to-one model

- Implement threads in the kernel.
- Pros: When a thread blocks, the kernel can schedule another thread.
- Cons: Creating and destroying threads are more expensive; trick for forks & signals

Many-to-many model

- Hybrid implementations.
- great, only complex to implement

Thread programming

POSIX threads (`man 7 pthreads`)

Description	Process	Thread
Process/thread identification.	`pid_t`	`pthread_t`
Get process/thread ID.	`getpid()`	`pthread_self()`
Create a new process/thread.	`fork()`	`pthread_create()`
Terminate the calling process/thread.	`exit()`	`pthread_exit()`
Wait for a specific process/thread to exit.	`waitpid()`	`pthread_join()`
Send a signal to a process/thread.	`kill()`	`pthread_kill()`
Release the CPU to let another process/thread run.	`sched_yield()`	`pthread_yield()`

Thread programming
Mutual exclusion

A mutex object of type `pthread_mutex_t` is similar to a binary semaphore.

Description	pthread mutex	Equivalent semaphore
Initialize a mutex.	`pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`	`semaphore mutex = 1;`
Lock a mutex. (Enter critical section.)	`pthread_mutex_lock(&mutex);`	`down(&mutex);`
Unlock a mutex. (Exit critical section.)	`pthread_mutex_unlock(&mutex);`	`up(&mutex);`

Note: `pthread_mutex_t` cannot be used as counting semaphores.

Thread programming
Semaphores

A semaphore object of type `sem_t` is similar to a counting semaphore.

Description	POSIX semaphore	Equivalent CS202 semaphore
Initialize a semaphore.	`sem_t s;` `sem_init(&s, 0, initial_value);`	`semaphore s = initial_value;`
Down a semaphore.	`sem_wait(&s);`	`down(&s);`
Up a semaphore.	`sem_post(&s);`	`up(&s);`
Destroy a semaphore.	`sem_destroy(&s);`	

Thread programming
Condition variables

Although semaphores provide a convenient and effective synchronization mechanism, using them incorrectly can cause hard-to-detect timing errors.

The `pthread` library provides another synchronization mechanism: **condition variables** (of type `pthread_cond_t`).

- Later, we'll see that semaphores can be easily implemented using mutexes and condition variables.

Condition variables allow a thread to wait for a condition to become true.

Example: the producer-consumer problem.

- If the buffer is full, the producer needs to block and be awakened when the buffer is not full.
- If the buffer is empty, the consumer needs to block and be awakened when the buffer is not empty.

Thread programming

Condition variables

Usage	Description
`pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`	Initialize a condition variable.
`pthread_cond_wait(&cond, &mutex);` Preventing deadlocks: no "hold-and-wait."	Wait on a condition. The calling thread must have locked the mutex. • When the thread is about to block, mutex will be unlocked automatically and atomically. • When the thread is unblocked, mutex will be locked again, automatically and atomically.
`pthread_cond_signal(&cond);`	Signal a condition. • If some threads are blocked on cond, at least one thread will be unblocked. • If no thread is blocked on cond, the signal is lost.

A thread pool is a design pattern for achieving concurrency of execution.

It maintains a pool of worker threads waiting for tasks to be dispatched.

Some data structures and functions are designed for single-threaded execution.

- Example: strtok() uses a static buffer while parsing a string, thus not thread safe.

One way to achieve thread safety is to make the function reëntrant.