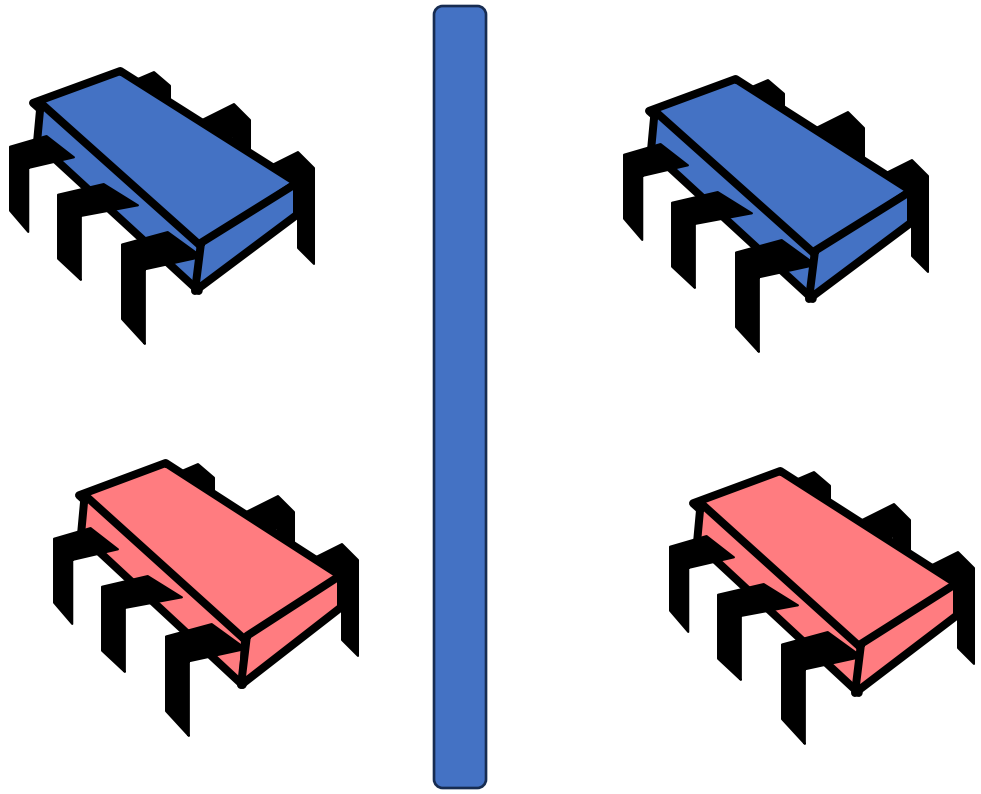
CSE113: Parallel Programming

- **Topics**:
  - Barriers
  - Processes



Announcements

- HW 4 grades will be released this week (after the holidays).

Announcements

- HW 5 is due this week on Thursday.

Announcements

SETs are out, please do them! It helps us out a lot.

Review

Barriers

Schedule

- **Barriers**
  - Specification
  - **Implementation**

Barrier Implementation

- First attempt at implementation

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        // ??
     }

}
```

Barrier Implementation

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        // What next?
     }

}
```

Barrier Implementation

First handle the case where the thread is the last thread to arrive

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        // What next?
     }

}
```

Barrier Implementation

Spin while there is a thread waiting at the barrier

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
     }

}
```

Barrier Implementation

Spin while there is a thread waiting at the barrier

Does this work?

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
     }

}
```

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

num_threads == 2

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

arrival_num = 1

arrival_num = 0

```
num_threads == 2
counter == 2
```

Thread 0:

```
B.barrier();
B.barrier();
```

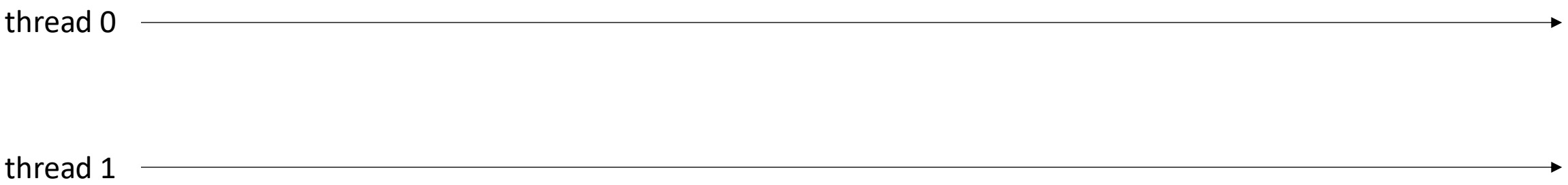
```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

arrival_num = 1

arrival_num = 0



num_threads == 2
counter == 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
            counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

arrival_num = 1

arrival_num = 0

```
num_threads == 2
counter == 0
```

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

Leaves barrier

arrival_num = 0

in a perfect world,
thread 1 executes now and leaves the barrier

thread 0 ⟶

thread 1 ⟶

```
num_threads == 2
counter == 0
```

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

Leaves barrier

arrival_num = 0

in a perfect world,
thread 1 executes now and leaves the barrier

but what if the OS preempted thread 1? Or it was asleep?

```
num_threads == 2
counter == 0
```

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

enters next barrier

arrival_num = 0

in a perfect world,
thread 1 executes now and leaves the barrier

but what if the OS preempted thread 1? Or it was asleep?

num_threads == 2
counter == 1

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

arrival_num == 0

arrival_num = 0

in a perfect world,
thread 1 executes now and leaves the barrier

but what if the OS preempted thread 1? Or it was asleep?

num_threads == 2
counter == 1

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

Thread 1 wakes up! Doesn't think its missed anything

arrival_num == 0

arrival_num = 0

in a perfect world,
thread 1 executes now and leaves the barrier

num_threads == 2
counter == 1

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

Thread 1 wakes up! Doesn't think its missed anything

arrival_num == 0

arrival_num = 0

in a perfect world,
thread 1 executes now and leaves the barrier

Both threads get stuck here!

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

Ideas for fixing?

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

Ideas for fixing?

Two different barriers that alternate?

Thread 0:

```
B0.barrier();
B1.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

```
B0.barrier();
B1.barrier();
```

Ideas for fixing?

Two different barriers that alternate?

Pros: simple to implement

Cons: user has to alternate barriers

Thread 0:

```
B0.barrier();
B1.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

```
B0.barrier();
B1.barrier();
```

Ideas for fixing?

Two different barriers that alternate?

Pros: simple to implement

Cons: user has to alternate barriers

```
B.barrier();
if (...) {
  B.barrier();
}
B.barrier();
```

How to alternate these calls?
Switching cannot be static,
has to be dynamic.

Sense Reversing Barrier

- Alternating “sense” dynamically

Thread 0:

```
B.barrier();
B.barrier();
```

sync on sense = false

Thread 1:

```
B.barrier();
B.barrier();
```

Sense Reversing Barrier

- Alternating “sense” dynamically

Thread 0:

```
B.barrier();
B.barrier();
```

sync on sense = true

Thread 1:

```
B.barrier();
B.barrier();
```

```
class SenseBarrier {
  private:
    atomic_int counter;
    int num_threads;
    atomic_bool sense;
    bool thread_sense[num_threads];
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
      sense = false;
      thread_sense = {true, ...};
    }

     void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
}
```

Set sense to what threads are waiting for

thread_sense = true

```
num_threads == 2
counter == 0
sense = false
```

thread_sense = true

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

Thread 1:

```
B.barrier();
B.barrier();
```

num_threads == 2
counter == 2
sense = false

thread_sense = true
arrival_num = 1

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = true
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

num_threads == 2
counter == 2
sense = false

thread_sense = true
arrival_num = 1

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = true
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

```
num_threads == 2
counter == 0
sense = true
```

thread_sense = false

arrival_num = 1

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = true

arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

num_threads == 2
counter == 0
sense = true

thread_sense = false
arrival_num = ?

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

thread_sense = true
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

Remember the issue! Thread 1 went to sleep around this time and thread 0 went into the barrier again!

num_threads == 2
counter == 1
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

thread_sense = true
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

num_threads == 2
counter == 1
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

thread_sense = true
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

both are waiting!,
but thread 1 can leave

num_threads == 2
counter == 1
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = false
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

both are waiting!,
but thread 1 can leave

num_threads == 2
counter == 1
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

thread_sense = false
arrival_num = ?

Thread 1:

```
B.barrier();
B.barrier();
```

Thread 1 finishes the barrier

num_threads == 2
counter == 1
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

thread_sense = false
arrival_num = ?

Thread 1:

```
B.barrier();
B.barrier();
```

Goes into the second barrier

```
num_threads == 2
counter == 2
sense = true
```

thread_sense = false

arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = false

arrival_num = 1

Thread 1:

```
B.barrier();
B.barrier();
```

Goes into the second barrier

num_threads == 2
counter == 2
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = false
arrival_num = 1

Thread 1:

```
B.barrier();
B.barrier();
```

Goes into the second barrier

```
num_threads == 2
counter == 0
sense = false
```

thread_sense = false

arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = false

arrival_num = 1

Thread 1:

```
B.barrier();
B.barrier();
```

Goes into the second barrier

```
num_threads == 2
counter == 0
sense = false
```

thread_sense = false

arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = false

arrival_num = 1

Thread 1:

```
B.barrier();
B.barrier();
```

thread 0 can leave, thread 1 can leave and the barrier works as expected!

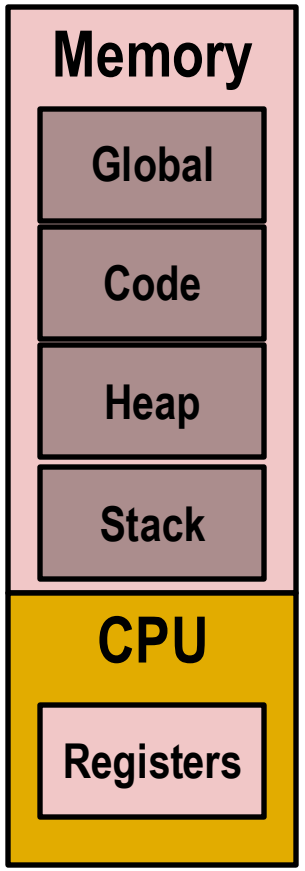
Processes

Processes

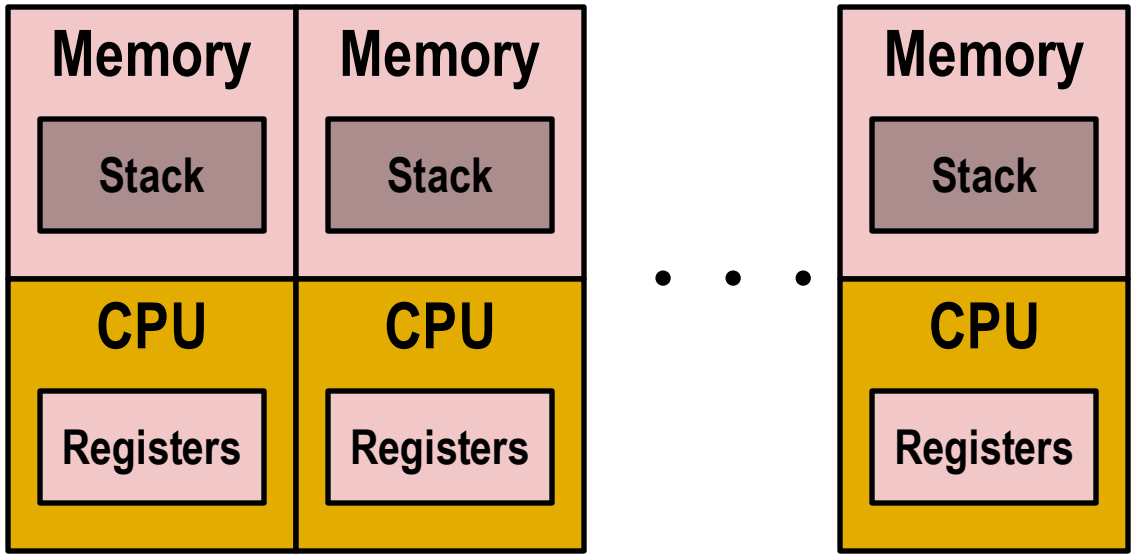
■**Definition: A *process* is an instance of a running program.**

■**Process provides each program with two key abstractions:**

- Logical control flow
  - Each program seems to have exclusive use of the CPU
- Private copy of program state
  - Register values (PC, stack pointer, general registers, condition codes)
  - Private virtual address space
    - Program has exclusive access to main memory
    - Including stack

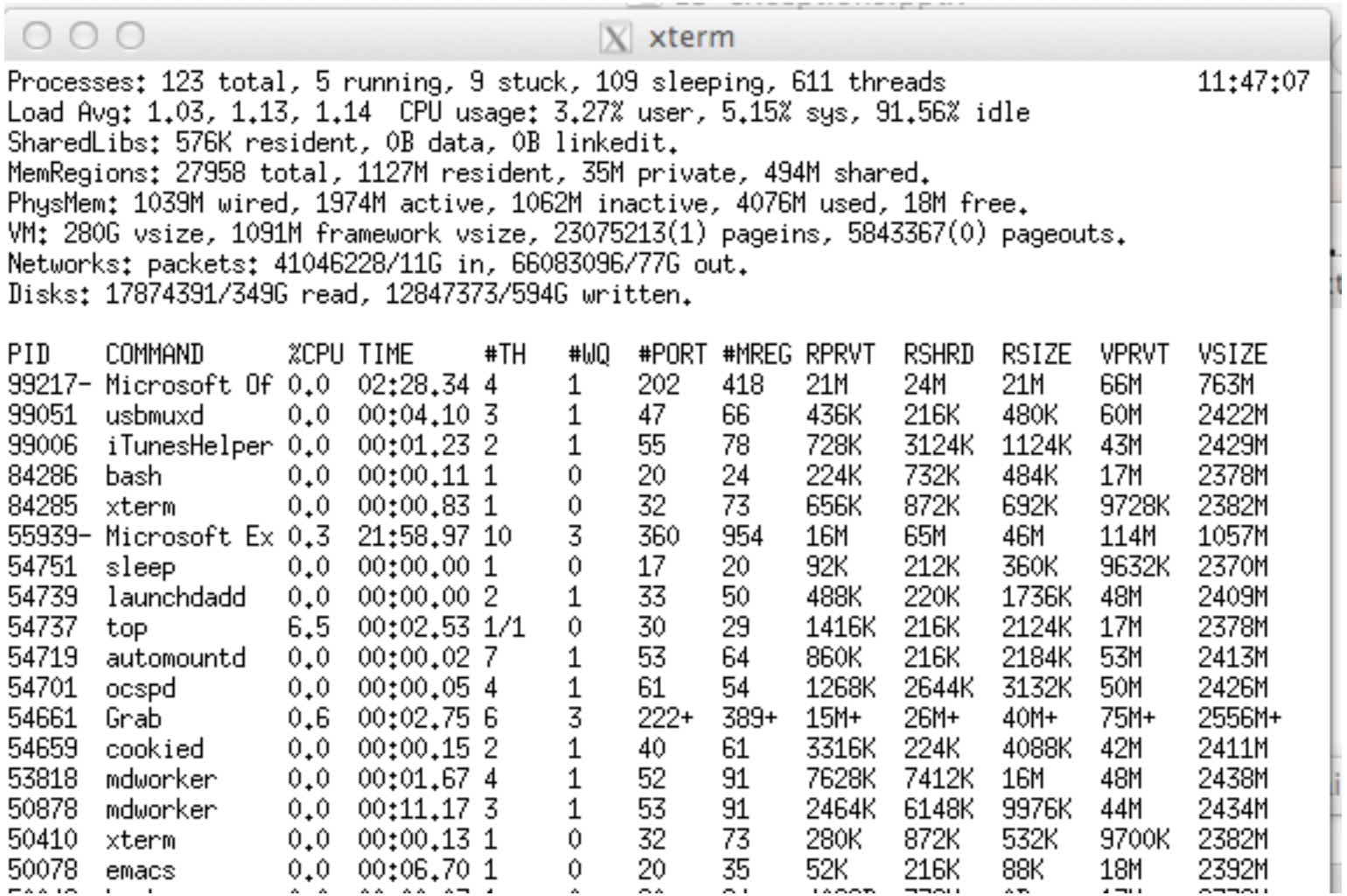


Multiprocessing: The Illusion



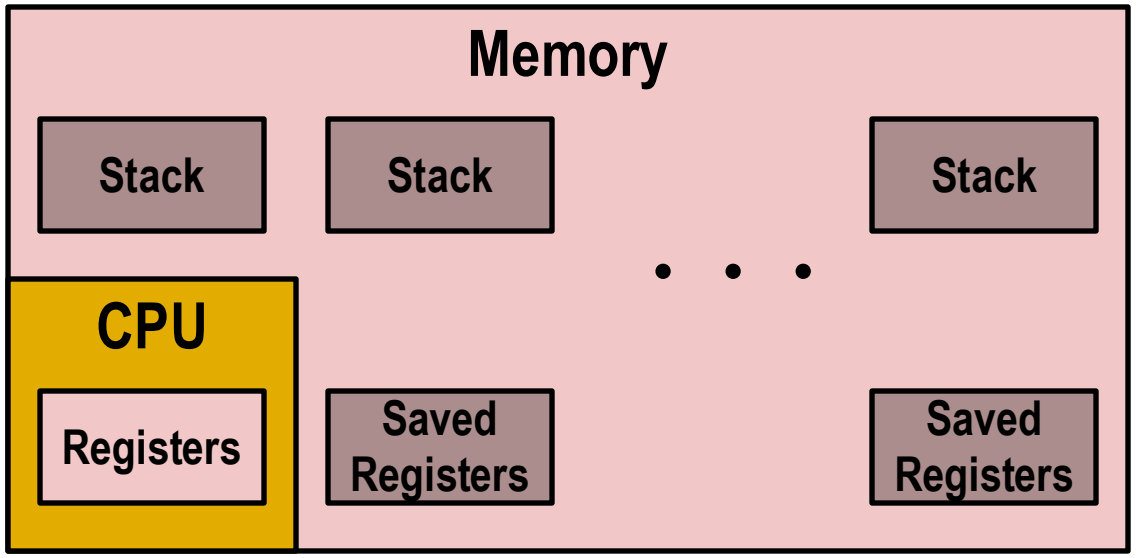
- **Computer runs many processes simultaneously**
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing Example



- **Running program "top" on Mac**
  - System has 123 processes, 5 of which are active
  - Identified by Process ID (PID)

Multiprocessing: The (Traditional) Reality

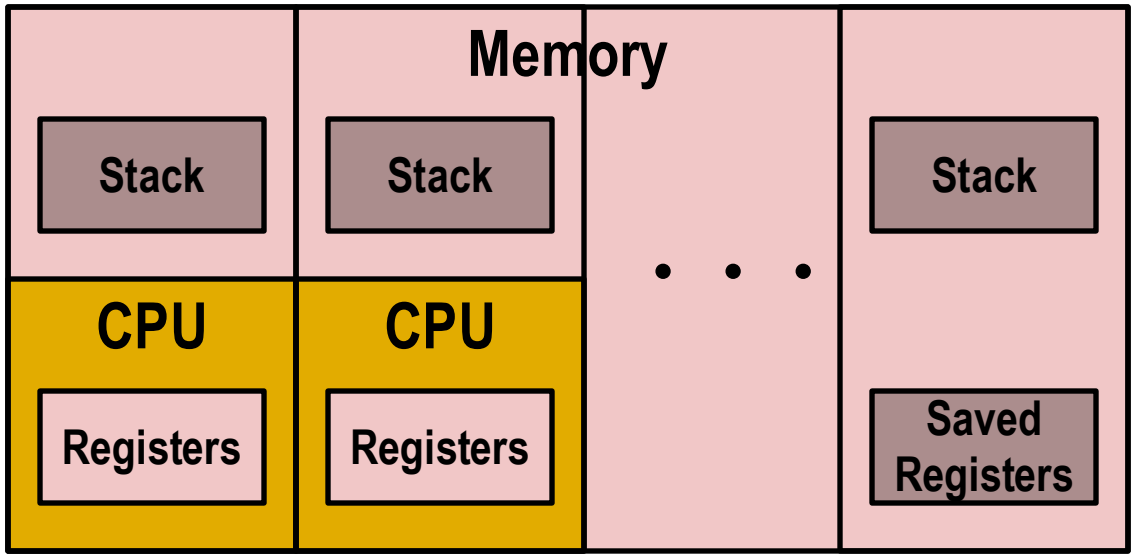


- **Single Processor Executes Multiple Processes Concurrently**
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system
  - Register values for non-executing processes saved in memory

The World of Multitasking

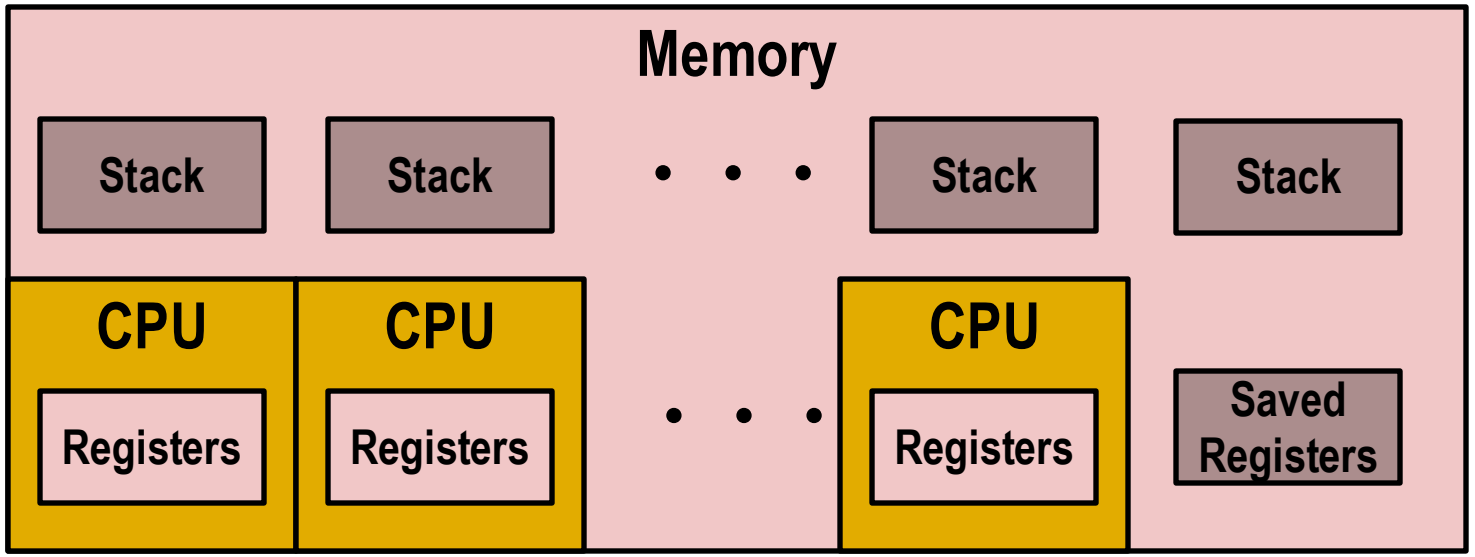
- **System runs many processes concurrently**
- **Regularly switches from one process to another**
  - Suspend process when it needs I/O resource or timer event occurs
  - Resume process when I/O available or given scheduling priority
- **Appears to user(s) as if all processes executing simultaneously**
  - Even though systems can only execute one process (or a small number of processes) at a time
  - Except possibly with lower performance than if running alone

Multiprocessing: The (New) Reality



- **Multicore processors**
  - Multiple CPUs on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Scheduling of processes onto cores done by OS

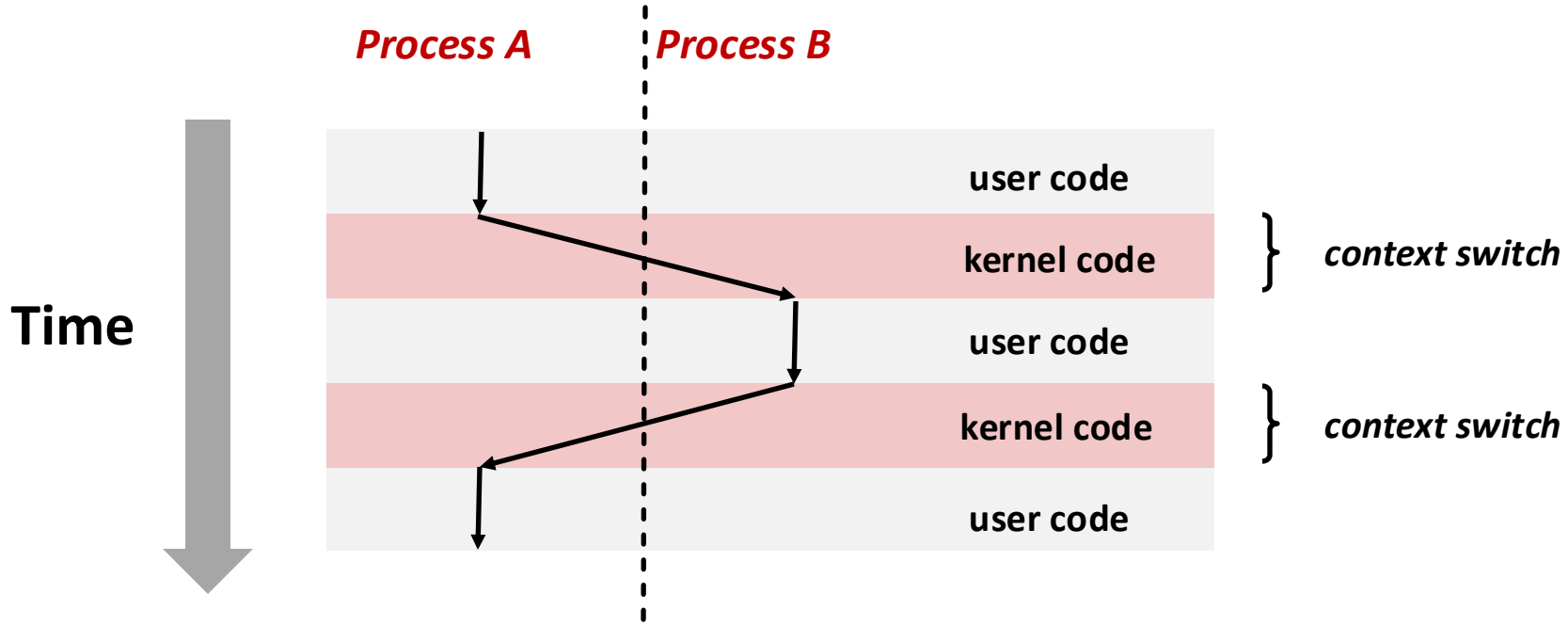
Multithreading: The Illusion



- **Single process runs multiple *threads* concurrently**
- **Each has own control flow and runtime state**
  - But view part of memory as shared among all threads
  - One thread can read/write the state of another

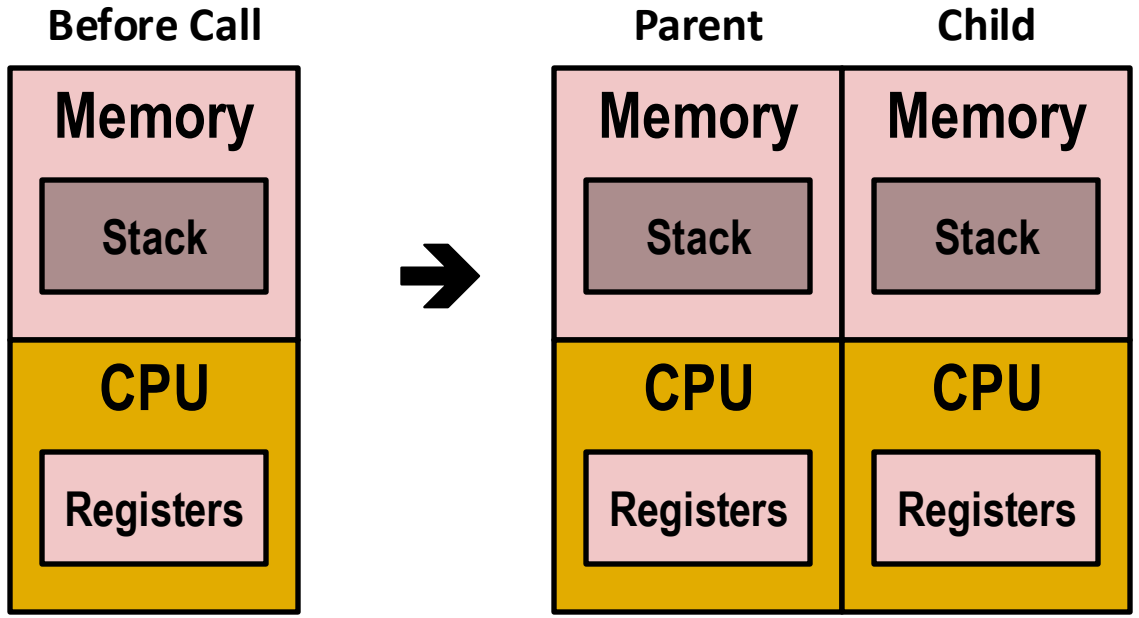
Context Switching

- Processes are managed by a shared chunk of OS code
- called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some user process
- Control flow passes from one process to another via a *context switch*



`fork`: Creating New Processes

- `int fork(void)`
  - creates a new process (child process) that is identical to the calling process (parent process)
  - (Appears to) create complete new copy of program state
  - Child & parent then execute as independent processes
    - Writes by one don't affect reads by other
    - But … share any open files



`fork`: Details

- **`int fork(void)`**
  - creates a new process (child process) that is identical to the calling process (parent process)
  - returns 0 to the child process
  - returns child's **`pid`** (process id) to the parent process

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

- **Fork is interesting (and often confusing) because**
- **it is called *once* but returns *twice***

Understanding fork

Process n

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Child Process m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

`hello from parent` *Which one is first?* `hello from child`

Fork Example #1

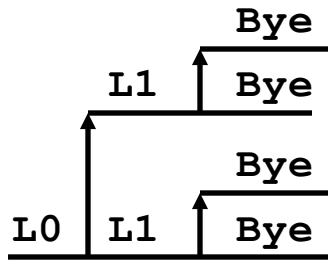
- **Parent and child both run the same code**
  - Distinguish parent from child by return value from **`fork`**
- **Start with same state, but each has private copy**
  - Including shared output file descriptor
  - Relative ordering of their print statements undefined

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

Fork Example #2

- Two consecutive forks

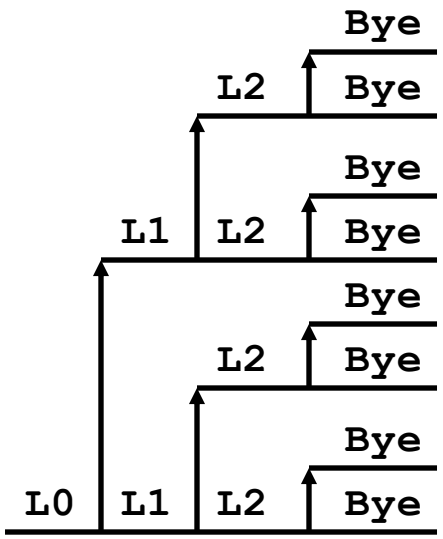
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #3

- Three consecutive forks

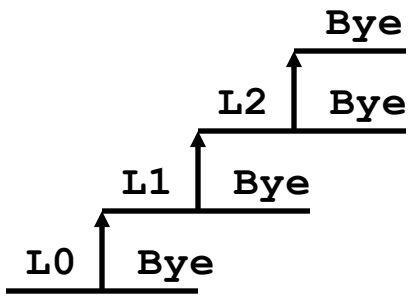
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



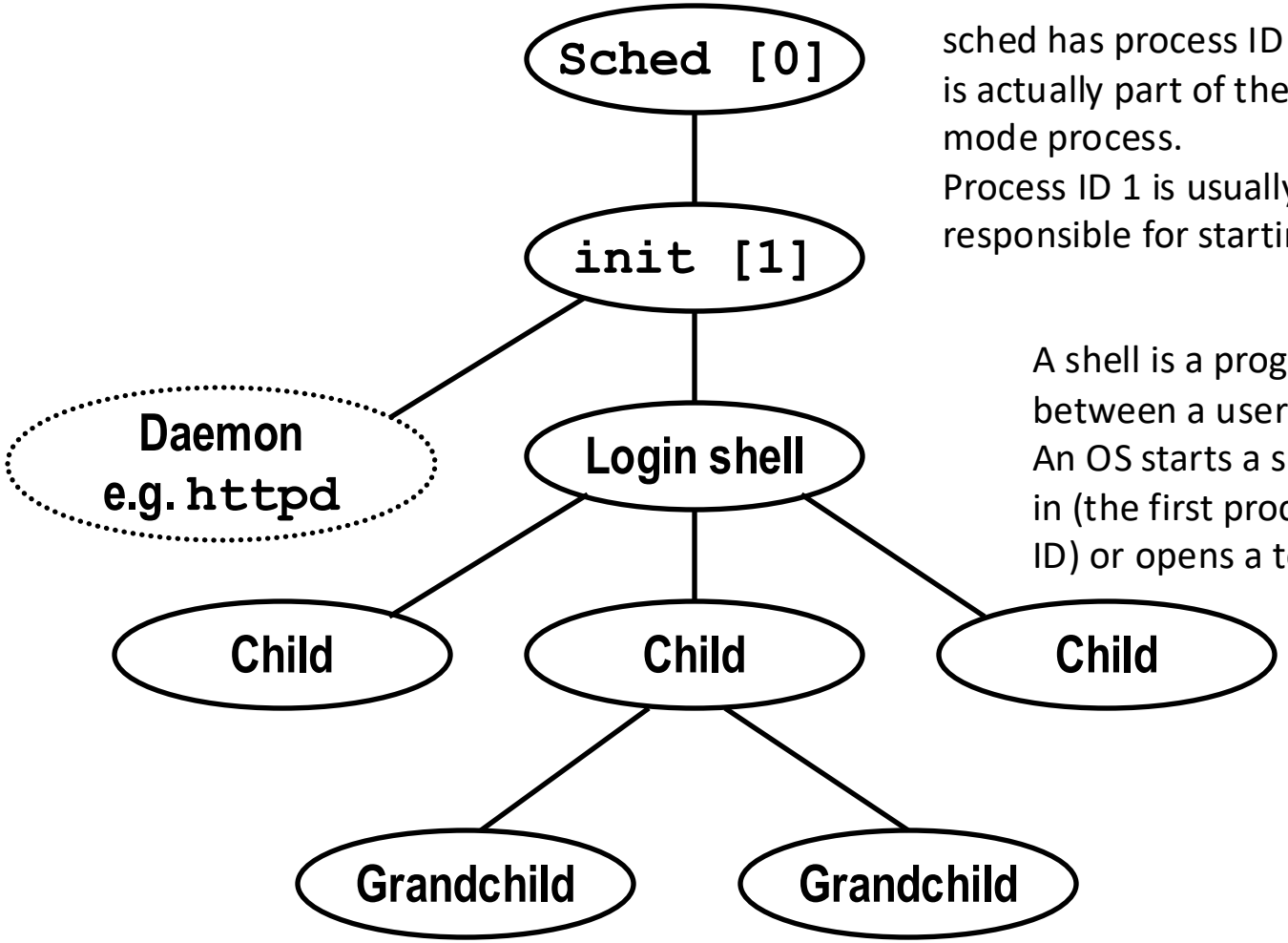
Fork Example #4

- Nested forks in children

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
       printf("L1\n");
       if (fork() == 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



Unix Process Hierarchy



sched has process ID 0 and is responsible for paging, and is actually part of the kernel rather than a normal user-mode process.
Process ID 1 is usually the init process primarily responsible for starting and shutting down the system.

A shell is a program that provides an interface between a user and an operating system (OS) kernel. An OS starts a shell for each user when the user logs in (the first process that executes under your user ID) or opens a terminal or console window.

A daemon is a computer program that runs as a background process, rather than being under the direct control of an interactive "user". Traditionally, the process names of a daemon end with the letter d

`exit`: Ending a process

- **`void exit(int status)`**
  - exits a process
    - Normally return with status 0
  - **`atexit()`** registers functions to be executed upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```