The Process Concept
The Process Concept

Process – a program in execution

Program
- description of how to perform an activity
- instructions and static data values

Process
- a snapshot of a program in execution
- memory (program instructions, static and dynamic data values)
- CPU state (registers, PC, SP, etc)
- operating system state (open files, accounting statistics etc)
Process Address Space

Each process runs in its own virtual memory *address space* that consists of:

- *Stack space* – used for function and system calls
- *Data space* – variables (both static and dynamic allocation)
- *Instructions* – the program code (usually read only)

Invoking the same program multiple times results in the creation of multiple distinct address spaces
Switching Among Processes

Program instructions operate on operands in memory and (temporarily) in registers.
Switching Among Processes

Saving all the information about a process allows a process to be *temporarily suspended* and later *resumed*.

![Diagram showing process switching]

OS suspends Prog1
Switching Among Processes

Saving all the information about a process allows a process to be *temporarily suspended* and later *resumed*.
Switching Among Processes

Program instructions operate on operands in memory and in registers

- Load A1, R1
- Load A2, R2
- Sub R1, R2, R3
- Store R3, A3
- ...

Prog2 has CPU
Prog1 is suspended
The Process Abstraction

Multiprogramming of four programs in the same address space
Conceptual model of 4 independent, sequential processes
Only one program active at any instant
The Scheduler

Processes

\[
\begin{array}{cccc}
0 & 1 & \ldots & n-2 \\
\hline
\text{Scheduler} & n-1 \\
\end{array}
\]

Lowest layer of process-structured OS
- handles interrupts & scheduling of processes
Sequential processes only exist above that layer
Process States

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available
How Are Processes Created?

Events that cause process creation:

- System startup
- Initiation of a batch job
- User request to create a new process
- Program request to create a new process

All result in the execution of a process creation system call (fork) from another process
Process Hierarchies

Parent process creates/forks child process
- each process is assigned a unique identifying number or process ID (PID)
- system calls for communicating with and waiting for child processes

Child processes can create their own child processes
How Do Processes Terminate?

Conditions that terminate processes:
- Normal exit (voluntary)
- Error exit (voluntary)
- Fatal error (involuntary)
- Killed by another process (involuntary)

All of the above are system calls!
Process Creation in UNIX

All processes have a unique process id

getpid(), getppid() system calls allow processes to get their information

Process creation

fork() system call creates a copy of a process and returns in both processes (parent and child), but with a different return value
exec() replaces an address space with a new program

Process termination, signaling

signal(), kill() system calls allow a process to be terminated or have specific signals sent to it
Process Creation Example

csh (pid = 22)

... pid = fork()
if (pid == 0) {
    // child...
    ...
    exec();
}
else {
    // parent
    wait();
}
...
Process Creation Example

csh (pid = 22)

```c
...  
  pid = fork();
  if (pid == 0) {
    // child...
    ...
    exec();
  }
  else {
    // parent
    wait();
  }
...```

csh (pid = 24)

```c
...  
  pid = fork();
  if (pid == 0) {
    // child...
    ...
    exec();
  }
  else {
    // parent
    wait();
  }
...```
Process Creation Example

csh (pid = 22)

...  
  pid = fork()
  if (pid == 0) {
    // child...
      ...
      exec();
    }
  else {
    // parent
    wait();
  }
...  

csh (pid = 24)

...  
  pid = fork()
  if (pid == 0) {
    // child...
      ...
      exec();
    }
  else {
    // parent
    wait();
  }
...
Process Creation Example

csh (pid = 22)

```c
...  
    pid = fork();
    if (pid == 0) {
        // child...
        ...
        exec();
    }
    else {
        // parent 
        wait();
    }
...  
```

csh (pid = 24)

```c
...  
    pid = fork();
    if (pid == 0) {
        // child...
        ...
        exec();
    }
    else {
        // parent 
        wait();
    }
...  
```
Process Creation Example

csh (pid = 22)

... pid = fork();
if (pid == 0) {
   // child...
   ...
   exec();
}
else {
   // parent
   wait();
}
...

ls (pid = 24)

//ls program
main(){
   //look up dir
   ...
}

Process Creation (fork)

Fork creates a new process by *copying* the calling process.

The new process has its own:
- Memory address space (copied from parent)
  - Instructions (same program as parent!)
  - Data
  - Stack
- Register set (copied from parent)
- Process table entry in the OS
Multiprocessors

How can we execute concurrent programs on multiprocessors using processes?

It would be nice to harness multiple CPUs in the execution of a single program instance.

Processes are a poor match for this.
Threads

Processes have the following components:

- an address space
- a collection of operating system state
- a CPU context ... a flow or thread of control

To use multiple CPUs on a multiprocessor system, a process would need several CPU contexts

- Thread fork creates new thread not memory space
- Multiple threads of control could run in the same memory space
- Can even be used in a single CPU system!
Threads

Threads share a process address space with zero or more other threads.

Threads have their own CPU context:
- PC, SP, register state,
- Stack

A traditional process could be viewed as a memory address space with a single thread.
Single Thread in Address Space

User Address Space

- Stack
  - routine1
  - var1()
  - var2()

- Text
  - main()
  - routine1()
  - routine2()

- Data
  - arrayA
  - arrayB

- Heap

Stack Pointer
- Program Counter
- Registers

Process ID
- Group ID
- User ID

Files
- Locks
- Sockets
Multiple Threads in Address Space
What Is a Thread?

A thread executes a stream of instructions
  - it is an abstraction for a control-flow

Practically, it is a processor context and stack
  - Allocated a CPU by a scheduler
  - Executes in a memory address space
Private Per-Thread State

Things that define the state of a particular flow of control in an executing program

- Stack (local variables)
- Stack pointer
- Registers
- Scheduling properties (i.e., priority)

The memory address space is shared with other threads in that process
Concurrent Access to Shared State

Changes made to shared state by one thread will be visible to the others!

Reading and writing memory locations requires synchronization

This is a major topic for later!
Programming With Threads

Split program into routines to execute in parallel
- True or pseudo (interleaved) parallelism

Alternative strategies for executing multiple routines
Why Use Threads?

Utilize multiple CPU’s concurrently

Low cost communication via shared memory

Overlap computation and blocking on a single CPU
  - Blocking due to I/O
  - Computation and communication

Handle asynchronous events
Processes vs Threads

GET / HTTP/1.0

HTTPD

disk
Processes vs Threads

Why is this not a good web server design?
Processes vs Threads

GET / HTTP/1.0

HTTPD

HTTPD

disk
Processes vs Threads

GET / HTTP/1.0

HTTPD

GET / HTTP/1.0

disk
Processes vs Threads

[Diagram showing the relationship between HTTPD and GET/HTTP/1.0 requests to the disk]
Pthreads: A Typical Thread API

Pthreads: POSIX standard threads
First thread exists in main(), creates the others

`pthread_create (thread, attr, start_routine, arg)`
- Returns new thread ID in “thread”
- Executes routine specified by “start_routine” with argument specified by “arg”
- Exits on return from routine or when told explicitly
Pthreads (continued)

pthread_exit (status)
- Terminates the thread and returns “status” to any joining thread

pthread_join (threadid, status)
- Blocks the calling thread until thread specified by “threadid” terminates
- Return status from pthread_exit is passed in “status”
- One way of synchronizing between threads

pthread_yield ()
- Thread gives up the CPU and enters the run queue
Using Create, Join and Exit

Master Thread

Worker Thread

Worker Thread

pthread_create() -> pthread_exit()

DO WORK

pthread_join()
An Example Pthreads Program

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    printf("%d: Hello World!\n", threadid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0; t<NUM_THREADS; t++)
    {
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc)
        {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Program Output

Creating thread 0
Creating thread 1
0: Hello World!
1: Hello World!
Creating thread 2
Creating thread 3
2: Hello World!
3: Hello World!
Creating thread 4
4: Hello World!

For more examples see: http://www.llnl.gov/computing/tutorials/pthreads
User-level threads

The idea of managing multiple abstract program counters above a single real one can be implemented without using privileged instructions

- Threads can be implemented in the OS or at user level

User level thread implementations

- Thread scheduler runs as user code (thread library)
- Manages thread contexts in user space
- The underlying OS sees only a traditional process above
Kernel-Level Threads

OS threads:
Thread-switching code is in the kernel
User-Level Threads Package

User-level threads:
The thread-switching code is in user space
Implementing Threads

We need a CPU in order to run a kernel thread.

We need a kernel thread or process in order to run a user-level thread.

We need a thread control block (TCB) structure in memory to hold thread state.

   It can be in user memory for user-level threads.
Implementing Threads

When a thread is created, what needs to happen?
When a thread exits, what needs to happen?
What will cause a thread switch to occur?
What will happen when a thread switch occurs?
Is the kernel really needed?
User-level threads

Advantages
- Cheap context switch costs among threads in the same process!
- Calls are procedure calls not system calls!
- User-programmable scheduling policy

Disadvantages
- How to deal with blocking system calls!
- How to overlap I/O and computation!
Types of “Context Switch”

Process switch
Thread switch
Kernel thread switch
User-level thread switch
Interrupt?
Function call?